



ROBOTSKI VID

Priročnik za laboratorijske vaje

Žiga Špiclin



Katalogni zapis o publikaciji (CIP) pripravili v Narodni in univerzitetni knjižnici v Ljubljani

COBISS.SI-ID 232295427

ISBN 978-961-243-479-3 (PDF)

URL: <https://lit.fe.uni-lj.si/gradivo/RobotskiVid-PrirocnikZaVaje.pdf>

Copyright © 2025 Založba FE. All rights reserved. Razmnoževanje (tudi fotokopiranje) dela v celoti ali po delih brez predhodnega dovoljenja Založbe FE prepovedano.

Naslov: Robotski vid; Priročnik za laboratorijske vaje

Recenzenta: prof. dr. Miran Bürmen, prof. dr. Tomaž Vrtovec

Založnik: Založba FE, Ljubljana

Izdajatelj: Fakulteta za elektrotehniko, Ljubljana

Urednik: prof. dr. Sašo Tomažič

Kraj in leto izida: Ljubljana, 2025

1. elektronska izdaja



Kazalo

1	Uvod	15
1.1	Robotski vid	15
1.1.1	Kratka zgodovina	17
1.1.2	Aplikacije	18
1.2	Uporaba priročnika	19
1.2.1	Minimalne strojne zahteve	20

I

Obdelava digitalnih slik

2	Upravljanje in prikazovanje slik	25
2.1	Python knjižnica <code>numpy</code>	25
2.2	Python knjižnica <code>pillow</code>	26
2.3	Python knjižnica <code>matplotlib</code>	26
2.4	Vaje z rešitvami	26
2.5	Naloge in vprašanja	32
3	Parametri kakovosti slik	35
3.1	Referenčni objekti in parametri kakovosti	36
3.2	Histogram slike	36
3.3	Vaje z rešitvami	37
3.4	Naloge in vprašanja	47

4	Preslikave sive in barv	49
4.1	Sivinske preslikave	49
4.2	Preslikave med barvnimi prostori	49
4.3	Vaje z rešitvami	50
4.4	Naloge in vprašanja	59
5	Osnovna obdelava slik	61
5.1	Filtriranje slik z diskretno konvolucijo	61
5.2	Interpolacija slik	61
5.3	Decimacija slik	62
5.4	Vaje z rešitvami	62
5.5	Naloge in vprašanja	69
6	Robustno iskanje objektov	73
6.1	Gradient sivinske slike	73
6.2	Harrisov detektor oglišč	74
6.3	Cannyev detektor robov	75
6.4	Houghov algoritem	76
6.5	Konvolucijske nevronske mreže	77
6.6	Vaje z rešitvami	78
6.6.1	Gradient sivinske slike	78
6.6.2	Harrisov detektor oglišč	79
6.6.3	Cannyev detektor robov	84
6.6.4	Zaznavanje premic s Houghovo preslikavo	89
6.6.5	Konvolucijske nevronske mreže	89
6.7	Naloge in vprašanja	92
6.7.1	Gradient sivinske slike	92
6.7.2	Harrisov detektor oglišč	92
6.7.3	Cannyev detektor robov	93
6.7.4	Zaznavanje krogov s Houghovo preslikavo	94
6.7.5	Konvolucijske nevronske mreže	95
7	Prostorske preslikave in poravnave	97
7.1	Geometrijske preslikave	97
7.2	Poravnava pripadajočih parov točk	98
7.3	Vaje z rešitvami	99
7.4	Naloge in vprašanja	106

II

Aplikacije robotskega vida

8	Geometrijska kalibracija kamere	113
8.1	Geometrijski kalibri	113
8.2	Model preslikave prostor-kamera	113

8.3	Prileganje modela s slikami kalibra	114
8.4	Vaje z rešitvami	114
8.5	Naloge in vprašanja	124
9	Določanje poze objektov v prostoru	127
9.1	Projekcija iz 3D v 2D	127
9.2	Mera podobnosti	127
9.3	Optimizacijski postopek	129
9.3.1	Primer: optimizacija nelinearne funkcije	129
9.3.2	Optimizacija mere podobnosti za 3D togo preslikavo	129
9.4	Vaje z rešitvami	130
9.5	Naloge in vprašanja	145
10	Rekonstrukcija 3D oblik	149
10.1	Fotometrični stereo in Lambertov model	149
10.2	Izračun albeda in normal na površino objekta	150
10.3	Rekonstrukcija 3D oblike	150
10.4	Vaje z rešitvami	151
10.5	Naloge in vprašanja	158
11	Sledenje in analiza gibanja	161
11.1	Parametrizacija preslikave	161
11.2	Linearizacija nelinearne kriterijske funkcije	161
11.3	Zaprto-zančna posodobitev preslikave	162
11.4	Piramidna implementacija	162
11.5	Vaje z rešitvami	163
11.6	Naloge in vprašanja	172
12	Lokalizacija in mapiranje okolja	175
12.1	Algoritem stereo vida	176
12.1.1	Ocenjevanje esencialne matrike E	177
12.1.2	Določanje poze (R, T) iz esencialne matrike E	178
12.2	Lokalizacija in mapiranje okolja	178
12.2.1	Posplošitev na n -pogledov	179
12.3	Vaje z rešitvami	179
12.4	Naloge in vprašanja	194
13	Vizualna kontrola kakovosti	195
13.1	Sistem za razpoznavanje vzorcev	195
13.1.1	Razgradnja objektov zanimanja	195
13.1.2	Analiza objektov	196
13.1.3	Razvrščanje	198

13.2	Vrednotenje sistemov odločanja	199
13.2.1	Občutljivost in specifičnost	199
13.2.2	ROC krivulja in AUC metrika	200
13.3	Vaje z rešitvami	200
13.4	Naloge in vprašanja	210

III

Dodatek

A	Singularni razcep	215
A.1	Matematična formulacija	215
A.2	Povezava z lastnimi vrednostmi in lastnimi vektorji	216
A.3	Lastnosti singularnega razcepa	216
A.4	Geometrijska interpretacija	217
A.5	Reševanje predoločenih sistemov enačb	217
B	Aproksimacijska 3D toga preslikava	219
B.1	Toga preslikava v 3D	219
B.1.1	Homogena 3D toga preslikava	220
B.2	Izpeljava aproksimacijske preslikave	220
B.3	Aproksimacijska preslikava v računskih korakih	222
C	Razvojna okolja	225
C.1	Osnovna namestitve in konfiguracija Pythona	226
C.1.1	Namestitve Pythona	226
C.1.2	Preverjanje namestitve	226
C.2	Upravljanje Python razvojnih okolij s pip	227
C.2.1	Namen in uporaba requirements.txt	228
C.2.2	Izbor verzij knjižnic	228
C.3	Upravljanje Python razvojnih okolij s conda	229
C.4	Uporaba Docker virtualnega okolja	230
C.4.1	Razumevanje osnovnih Docker konceptov	231
C.4.2	Ustvarjanje virtualnega okolja	232
C.5	Uporaba Docker mape	235
C.5.1	Nastavitev Docker kontejnerja za daljinski dostop	236
C.6	Uporaba spletnih razvojnih okolij: Google Colab	237
C.6.1	Navodila za ustvarjanje in konfiguracijo projekta	238
C.6.2	Primer uporabe Google Colab	239
C.6.3	Namigi za uporabo GPU/TPU:	239
C.7	Lokalno razvojno okolje: Visual Studio Code	240
C.7.1	Namestitve in osnovna konfiguracija Visual Studio Code	240
C.7.2	Uporaba lokalne Python distribucije	241
C.7.3	Uporaba lokalnega Docker kontejnerja	242

D	Vodenje različic kode z Git	245
D.1	Kaj je Git in zakaj ga uporabljamo?	245
D.2	Namestitev Gita	245
D.2.1	Namestitev na Windows	245
D.2.2	Namestitev na macOS	246
D.2.3	Namestitev na Linux	246
D.2.4	Preverjanje namestitve	246
D.3	Osnovne nastavitve	246
D.3.1	Nastavitev uporabniškega imena in e-pošte	246
D.3.2	Privzeto nastavljanje urejevalnika	246
D.3.3	Prikaz konfiguracije	247
D.4	Ustvarjanje in kloniranje repozitorija	247
D.4.1	Inicializacija novega repozitorija	247
D.4.2	Kloniranje obstoječega repozitorija	247
D.5	Osnovni Git ukazi	248
D.5.1	Preverjanje statusa datotek	248
D.5.2	Dodajanje sprememb v pripravo	248
D.5.3	Potrjevanje sprememb	248
D.5.4	Ogled zgodovine sprememb	248
D.6	Uporaba vejitev	249
D.6.1	Ustvarjanje in preklapljanje vej	249
D.6.2	Združevanje vej	249
D.6.3	Reševanje konfliktov	250
D.7	Povezava z oddaljenim repozitorijem	250
D.7.1	Dodajanje oddaljenega repozitorija	250
D.7.2	Pošiljanje sprememb	251
D.7.3	Pridobivanje sprememb	251
D.8	Pogoste napake in koristni namigi	251
D.8.1	Razveljavljanje sprememb	252
D.8.2	Pridobivanje izgubljenih potrjenih sprememb	252
D.8.3	Ignoriranje datotek	252
D.9	Zaključek	253
D.10	Dobra praksa priprave Git repozitorija	253
D.11	Zaključek	253
D.11.1	Zakaj je Git ključen za razvijalce?	254
D.11.2	Nadaljnji viri za učenje	254
	Bibliografija	255
	Knjige	255
	Članki	256
	Konferenčni članki	256
	Razno	256
	O avtorju	259

*Samotu, Svitu
in Tanji* ■



Predgovor

Pričujoči priročnik predstavlja študijsko gradivo, navodila za vaje in domače naloge pri laboratorijskih vajah predmeta *Robotski vid* na univerzitetnem magistrskem študiju elektrotehnike druge stopnje na študijski smeri Robotika. Predmet *Robotski vid* med študenti velja za zahteven predmet. Dostopnost kakovostne študijske literature in gradiv lahko olajša obvladovanje snovi ter tako omili zahtevnost, kar želim doseči s pripravo pričujočega prostodostopnega priročnika. Nastal je iz posamičnih navodil za izvedbo laboratorijskih vaj pri tem predmetu v študijskih letih od 2012/2013 do 2024/2025. Predhodnik tega priročnika so zbrana gradiva za laboratorijske vaje naslovljena *Robotski vid: laboratorijske vaje v programskem jeziku Matlab* [9], na osnovi katerih so, kot pove naslov, izvajanja vaj temeljila na uporabi razvojnega okolja Matlab. Leta 2017 sem vaje prilagodil za uporabo programskega jezika Python, saj ponuja odprtokodno rešitev brez licenčnih omejitev, široko podporo skupnosti, bogate knjižnice za računalniški vid in strojno učenje (npr. OpenCV, TensorFlow) ter boljše prilagodljivost za sodobne aplikacije.

Namen priročnika je seznaniti študente s teorijo algoritmov obdelave in analize digitalnih slik, in podati jasna navodila in smernice za izvedbo laboratorijskih vaj ter dodatnih in/ali domačih nalog. Priročnik obsega 12 laboratorijskih vaj, ki študente seznanijo z uporabo programskega jezika Python in pripadajočih programskih knjižnic za specifične naloge in cilje. Laboratorijske vaje so organizirane v dva sklopa, in sicer na (i) vaje s področja obdelave in analize digitalnih slik in (ii) vaje s področja robotskega vida. Vaje s področja obdelave in analize digitalnih slik vključujejo upravljanje in prikazovanje slik, parametri kakovosti slik, preslikave sivin in barv, osnovna obdelava slik, robustno iskanje objektov in prostorske preslikave in poravnave slik. Na osnovi teh vaj študenti pridobijo temeljna znanja za kasnejše zahtevnejše laboratorijske vaje, kjer morajo ta znanja uporabiti in nadgraditi v aplikacije robotskega vida. Vaje s področja robotskega vida vključujejo pogoste, a raznolike aplikacije robotskega vida kot so geometrijska kalibracija kamere, določanje poze objektov v prostoru, rekonstrukcija 3D oblik, sledenje in analiza gibanja, lokalizacija in mapiranje okolja in vizualna kontrola kakovosti.

Zasnova laboratorijskih vaj temelji na principu *od spodaj navzgor* tako, da študentje tekom izvajanja ustvarjajo lastno programsko knjižnico za vse obravnavane postopke in algoritme. V kasnejših vajah posamezne časovno kritične algoritme ali dele algoritmov nadomestijo s klici

implementacij v uveljavljenih Python knjižnicah. Na ta način študenti podrobno spoznajo zasnovo in delovanje posameznih algoritmov ter njihovo programsko implementacijo, vpliv implementacije na pravilnost in časovno zahtevnost izvajanja in osvojijo razhroščevanje kode. Pomemben vidik uporabe principa *od spodaj navzgor* je izgradnja samozvesti za samostojno implementacijo zahtevnih algoritmov ter kritičnega mišljenja pri vrednotenju delovanja. Kljub temu, da bodo študenti pri morebitnem kasnejšem profesionalnem delu na tem področju praviloma uporabljali implementacije v uveljavljenih programskih knjižnicah kot je OpenCV, pa bodo gotovo usposobljeni za implementacijo morebitnih manjkajočih funkcij za povezovanje obstoječih implementacij v delujočo rešitev.

Poleg teorije in algoritmov obdelave in analize digitalnih slik ter aplikacij robotskega vida študenti tekom izvajanja laboratorijskih vaj osvojijo matematična in mnoga programska orodja. Za čim lažje osvajanje dela je v dodatku tega priročnika dodan sklop poglavij III, v katerih je podana razlaga singularnega razcep in njegova uporaba, toga 3D aproksimacijska poravnava, kratek pregled dela z orodji za upravljanje nameščanja Python knjižnic Conda in Docker in pregled razvojnih okolij kot so Visual Studio Code in Google Colab, ter kratek pregled dela z orodjem Git za arhiviranje in vodenje različic kode. Obvladovanje omenjenih orodij je nedvomno splošno uporabno pri reševanju inženirskih problemov.

Nastanek tega priročnika ne bi bil mogoč brez izjemnih sodelavcev, tako bivših kot sedanjih, iz Laboratorija za slikovne tehnologije na Fakulteti za elektrotehniko, Univeze v Ljubljani, ki so vsak po svoje prispevali k zasnovi in vsebini laboratorijskih vaj pri predmetu Robotski vid. Posebej se zahvaljujem recenzentoma prof. dr. Miranu Bürmenu in prof. dr. Tomažu Vrtovcu za natančen pregled osnutka tega priročnika in koristne komentarje in popravke. Iskreno zahvalo posvečam asistentom Jaki Kutrašniku, Timu Jermanu, Žigi Bizjaku, Lari Dular in Domnu Preložniku, ki so sodelovali oz. sodelujejo pri praktični izvedbi laboratorijskih vaj in so dodobra preverili pedagoško smiselnost podane teorije, navodil in smernic za izvedbo vaj kot dodatnih nalog in vprašanj.



1. Uvod

Ljudje z neverjetno lahkoto zaznavamo tridimenzionalno (3D) strukturo sveta okoli nas. Na primer, iz slike 1.1a bi vsak človek zlahka sklepal, da "bližnji moški, ki ga vidimo v hrbet, hodi proti levi strani ulice, široka tlakovana ulica rahlo zavije v desno približno 100 metrov od točke, kjer je bila slika posneta, gosta pozidava nakazuje, da je bila slika posneta v mestu, in na podlagi oblačil (vsi moški nosijo klobuk in temno obleko) se zdi, da izvira izpred več kot 100 let." Tovrstna opažanja je matematično nemogoče opisati. A kako pridemo do njih? Temeljijo na predpostavkah, predhodnem znanju in vsakodnevnih izkušnjah iz realnega sveta.

Kljub navidezni lahkotnosti, s katero zaznavamo svet okoli sebe, ostaja definicija človeškega vida še vedno izmuzljiva – celo za največje mislece v zgodovini. Razmislimo o naslednjih poskusih, da bi ga opredelili:

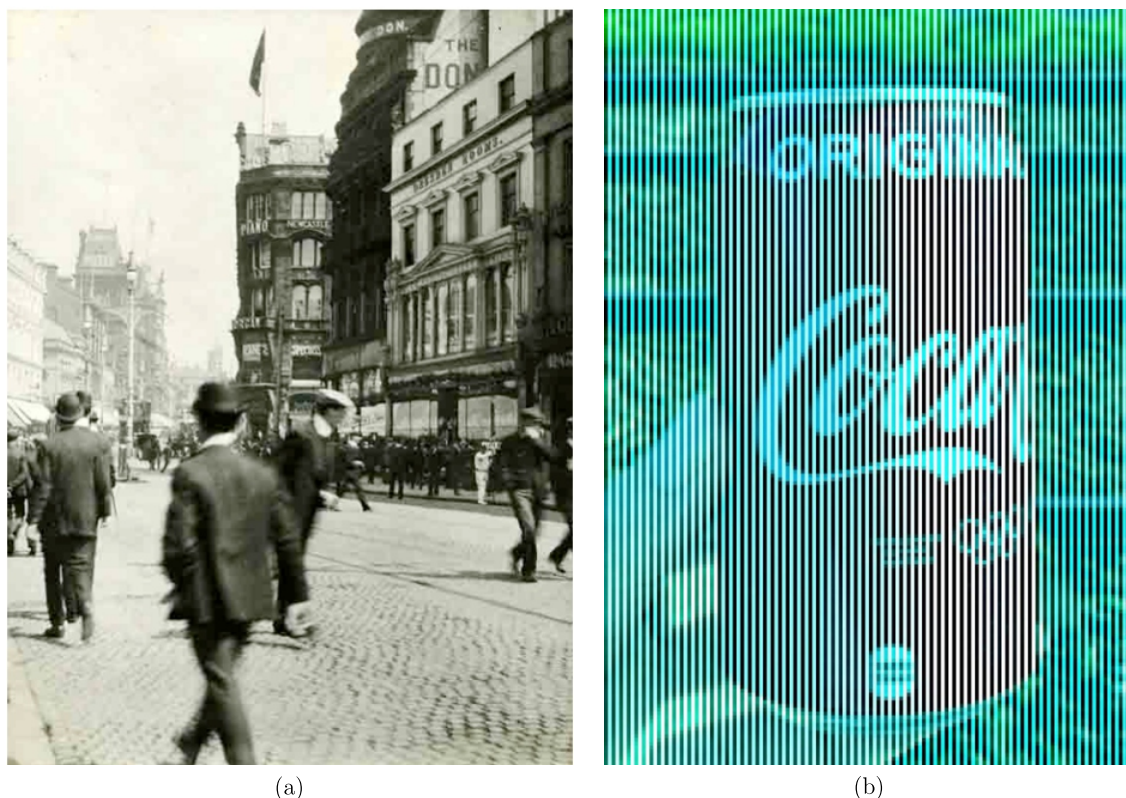
Citat 1.1 — Človeški vid.

"Vision is the act of knowing what is where by looking." — Aristotle (384 – 322 pr. n. št.) *"Vision is the act of unconscious inference."* — Hermann von Helmholtz (1821 – 1894)

Kar se sprva zdi preprost proces – odpreti oči in videti – se hitro zaplete v prepleteno mrežo nevroznanosti, fizike, psihologije, filozofije, umetne inteligence in kognitivne znanosti. Verjamete, da v sliki 1.1b ni niti pikice rdeče barve? Paradoksalno je, da bolj ko razkrivamo mehanizme svetlobe, nevronov in zaznavanja, bolj zabrisana postaja definicija človeškega vida. Če je bila sprva pomembno vprašanje *"ali veste, kaj je kje?"*, se je nadalje razvilo v razpravo o tem, *"kaj sploh pomeni vedeti?"*. Lahko morda bolj točno odgovorimo na vprašanje *"kaj je robotski vid?"*

1.1 Robotski vid

Robotski vid je področje, ki omogoča robotom zaznavanje, razumevanje in interpretacijo vizualnih informacij iz okolja, da se lahko ustrezno odzovejo. Robotski vid črpa znanja iz številnih sorodnih področij, kot je prikazano na sliki 1.2. Povezan je z različnimi znanstvenimi in inženirskimi disciplinami, saj združuje metode za zaznavo, interpretacijo in odzivanje na vizualne informacije. Tesno je povezan z računalniškim vidom (ang. *computer vision*), ki se ukvarja z obdelavo in



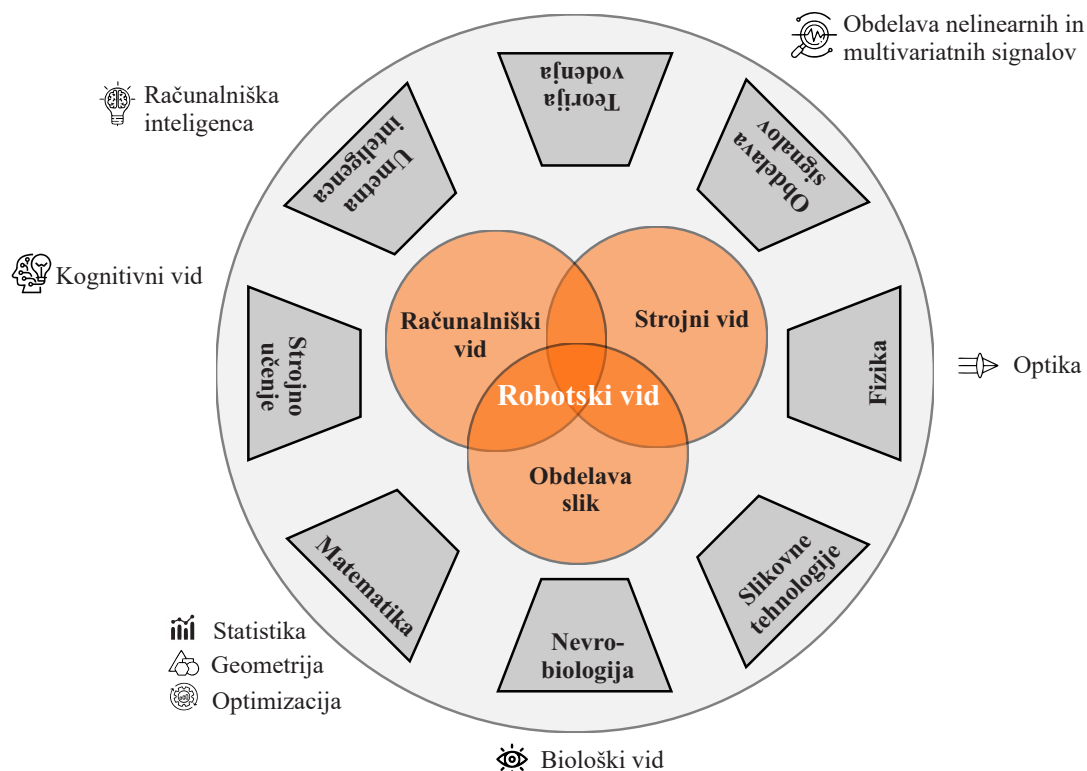
Slika 1.1: (a) Slika prizora iz preteklosti – kaj vidite na tej sliki? (b) Slika verjetno najbolj poznane pločevinke na svetu – na tej sliki **ni rdeče barve**, a zakaj potem vidimo rdečo pločevinko?

analizo slik ter videoposnetkov, vendar robotski vid vključuje tudi *dejanje* – torej uporabo zaznanih podatkov za fizično interakcijo robota z okoljem. Po drugi strani pa strojni vid (ang. *machine vision*) pogosto označuje industrijsko uporabo vizualnih sistemov, kjer so kamere in algoritmi namenjeni specifičnim nalogam, kot je kontrola kakovosti ali avtonomna proizvodnja. Tako lahko sklenemo, da je robotski vid posebna veja računalniškega vida, ki presega zgolj razumevanje slike in vključuje sposobnost odločanja ter delovanja v fizičnem svetu.

Z robotskim, računalniškim in strojnim vidom povezano temeljno teoretično področje je obdelava signalov, pri čemer obravnavamo slike in videje kot večdimenzionalne signale. Obdelava signalov se v splošnem osredotoča na preoblikovanje surovih podatkov v uporabne informacije. Fizikalni temelji robotskega vida izhajajo iz fizike, optike in slikovnih tehnologij, saj razumevanje svetlobe, leč, senzorjev in kamer vpliva na zajem in obdelavo slik.

Ker je človeški vid navdih za robotski vid, igra ključno vlogo tudi nevrobiologija in biologija vida, ki pomagata pri razumevanju, kako naravni sistemi zaznavajo in interpretirajo vizualne informacije. Matematične discipline, kot so geometrija, statistika in optimizacija, so osnova za algoritme, ki omogočajo modeliranje prostora, analizo oblik in izboljšanje računalniških metod za vid.

Robotski vid je močno povezan s strojnim učenjem, kognitivnim vidom, umetno inteligenco in računalniško inteligenco, ki omogočajo samodejno učenje vizualnih vzorcev in izboljšanje razumevanja slik. Nenazadnje sta robotsko upravljanje in teorija vodenja ključna elementa, ki omogočata, da sistem vida ni le pasiven opazovalec, ampak aktivni del avtonomnega robota, ki se na podlagi vizualnih informacij odloča in deluje v resničnem svetu.



Slika 1.2: Robotski vid in sorodna področja.

1.1.1 Kratka zgodovina

Začetek računalniške interpretacije slik vida sega v leto 1966, ko je profesor Marvin Minsky (MIT; 1927–2016) uvedel *računalniški vid* kot poletni dodiplomski projekt, pri čemer je svojemu študentu naročil: “”*tekom poletja poveži kamero z računalnikom in pripravi računalnik tako, da opiše, kar vidi kamera.*” Še danes bi bil to zelo zahteven (dodiplomski) projekt. Od tistega leta dalje se je razvoj računalniškega vida postopoma premikal od interpretacije sintetičnih prizorov do prepoznavanja kompleksnih vizualnih podatkov iz resničnega sveta.

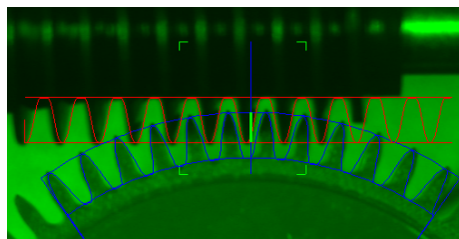
V grobem lahko za vsako desetletje identificiramo ključne dosežke in mejnike. V šestdesetih je bil poudarek na interpretaciji sintetičnih scen, saj je bil svet vektorjev in enostavnih geometrijskih oblik edini, ki ga je lahko računalnik razumel. Sedemdeseta so prinesla prvo interpretacijo izbranih resničnih slik, čeprav je bil postopek še vedno daleč od avtomatizacije. V osemdesetih je sledil vzpon umetne inteligence (AI; ang. *Artificial Intelligence*), vendar se je področje zaradi nezadostnih računskih zmogljivosti tedanjih računalnikov začelo nagibati k geometriji in matematičnemu modeliranju, ki sta omogočila robustne metode analize slik umetno ustvarjenih objektov, ki imajo dobro definirano obliko in druge lastnosti. V devetdesetih je prišlo do preboja v razpoznavanju obrazov, pri čemer je statistična analiza postala prevladujoča metodologija. V dvatisočih je računalniški vid dobil nov zagon z velikimi označenimi bazami slik (kot so ImageNet [17]) in prvimi uspešnimi metodami obdelave videa. Dvatisočdeseta so bila obdobje strojnega učenja (ML; ang. *Machine Learning*) in globokih nevronske mreže, ki so s svojimi zmogljivostmi začela presegati klasične algoritme. V dvatisočdvajsetih smo pričali vzponu temeljnih globokih modelov (ang. *foundational models*) in velikih jezikovnih modelov (ang. *large language models*), ki ne prepoznavajo le slik, ampak jih tudi razumejo, opisujejo in celo ustvarjajo. Ironično je, da se danes zdi Minskyjev študentski projekt skoraj otročje preprost—dokler ne vprašamo sodobnega računalnika, naj “*opredeli, kar vidi*”, in dobimo odgovor, ki je ali neverjetno natančen ali pa popolnoma absurd. “*Resnično izkušnjo sveta*” s svojo bogato kompleksnostjo težko matematično

opredelimo, definiramo, modeliramo in na koncu eksplicitno zakodiramo v rešitve na področju vida. Zato se moramo pogosto zanašati na fizikalne in verjetnostne modele, da razjasnimo med možnimi rešitvami danega problema.

1.1.2 Aplikacije

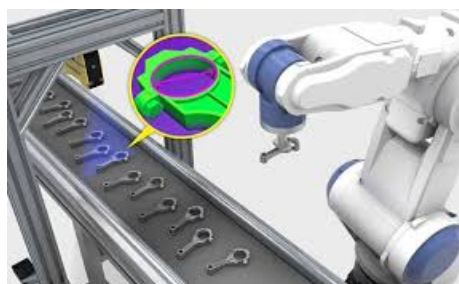
Tipične naloge robotskega vida pokrivajo širok spekter aplikacij, kjer mora robot vizualno zaznavati, analizirati in se odzvati na okolje. Nekatere ključne naloge, ki jih obravnavamo v tem priročniku vključujejo:

- **Geometrijska merjenja:** Z določitvijo notranjih (goriščna razdalja, popačenje leč) in zunanjih (položaj, orientacija) parametrov kamere zagotavlja točne meritve in robustnost v različnih pogojih. To omogoča geometrijska kalibracija kamere, s katerim natančno modeliramo preslikavo med 2D slikami in 3D svetom. Glej poglavje 8.



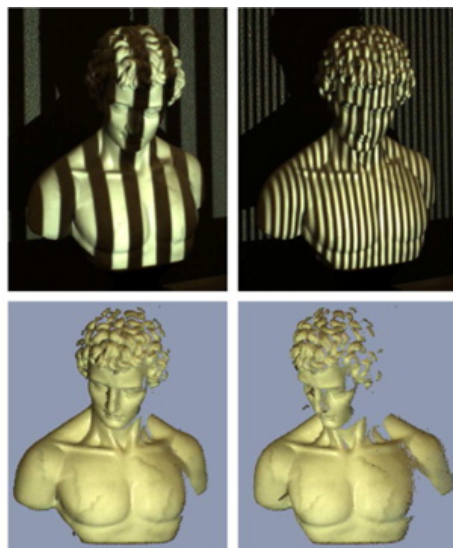
Slika 1.3: Geometrijske meritve ujemanja zobnikov.

- **Zaznavanje in pobiranje predmetov** (ang. *pick and place*): Roboti v proizvodnji ali skladiščih uporabljajo vid za natančno zaznavanje poze predmetov, kar omogoča njihovo pobiranje z robotskimi prijemači. Zanimiva sorodna aplikacija je določanje poze anatomskih struktur v medicinskih slikah, za namen slikovno podprtih posegov ali za vodenje kirurškega robota. Glej poglavje 9.



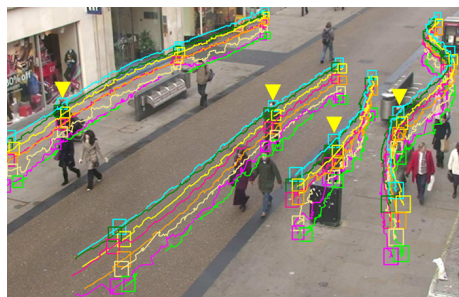
Slika 1.4: Zaznavanje in pobiranje kosov na tekočem traku [30].

- **Rekonstrukcija 3D oblike:** Omogoča pridobivanje 3D modelov objektov in okolja na podlagi 2D slik ali senzorskih podatkov. Primeri uporabe so (i) merjenje dimenzij izdelkov in preverjanje napak na proizvodni liniji, (ii) izboljšano prepoznavanje objektov, ocena razdalj in načrtovanje gibanja v prostoru, (iii) digitalizacija zgodovinskih artefaktov in arhitekturnih struktur omogoča ohranjanje kulturne dediščine ter virtualne rekonstrukcije izgubljenih objektov. Glej poglavje 10.



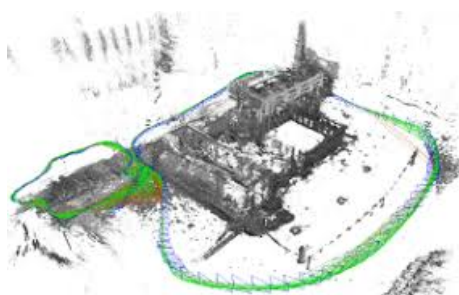
Slika 1.5: Strukturirana svetloba za 3D rekonstrukcijo kipa [27].

- **Sledenje premikajočim objektom:** Sistem robotskega vida lahko spremlja gibanje določenih objektov v realnem času, kar je uporabno pri robotih, ki lovijo premikajoče se tarče ali spremljajo delavce v sodelovalnih robotskih sistemih. Glej poglavje 11.



Slika 1.6: Sledenje gibanja oseb na ulici [26].

- **Lokalizacija in navigacija:** Robot s pomočjo kamer in senzorjev zaznava svojo okolico, gradi mapo okolja in prepozna ovire ter določa svojo pot v prostoru. To je ključnega pomena za avtonomne mobilne robote, droni in samovozeča vozila. Glej poglavje 12.



Slika 1.7: Mapa okolja in trajektorija kamere [23].

- **Preverjanje kakovosti objektov** (ang. *visual inspection*): Robotski vid omogoča prepoznavo različnih predmetov na podlagi barve, oblike ali texture, kar se uporablja v industriji za sortiranje izdelkov in avtomatizirano proizvodnjo, za odkrivanje napak v izdelkih, preverjanje skladnosti dimenzij ali odkrivanje nepravilnosti na površinah. Glej poglavje 13.



Slika 1.8: Vizualno preverjanje kakovosti svedra [29].

Uveljavljajo se tudi številne druge aplikacije robotskega vida. Velja, da z lahko malce domišljije in spretnosti v industrijskem ali pa domačem okolju katerokoli kamero razvijemo v inteligentni senzor, ki je prav tako aplikacija robotskega vida.

1.2 Uporaba priročnika

Namen tega priročnika je z vajami okrepiti spretnosti in veščine tako matematike, fizike, programiranja in uporabe programskih orodij in knjižnic tako, da bo uporabnik priročnika zmožen samostojne obravnave in reševanja problemov na področju robotskega vida. Vsako poglavje poda teoretično podlago, razgradi problem na posamezne podprobleme in nato skozi praktične primere blokov programske kode Python vodi uporabnika do rešitve posameznega podproblema, katerih rešitve nato sestavi v končno rešitev. Vsa koda iz vaje in pripadajoče rešitve ter primeri so kot sestavni del tega priročnika objavljeni na javnem Git repozitoriju:

- <https://gitlab.lst.fe.uni-lj.si/vaje/robotski-vid.git>

Priročnik se deli na tri dele. Prva dva dela podajata navodila za vaje z rešitvami in dodatne naloge

ter vprašanja. V delu I so vaje iz obdelave digitalnih slik in v delu II vaje iz specifičnih aplikacij robotskega vida. Vaje v priročniku so strukturirane po učnem načrtu, ki ga prikazuje slika 1.3. Vsaki vaji je pridruženo predavanje, ki zagotavlja širšo teoretično podlago posameznega področja vaje. Drsnice predavanj so prosto dostopne na spletu v .pdf obliki:

- [10] Žiga Špiclin. *Robotski vid: predavanja*. Ljubljana: Univerza v Ljubljani, Fakulteta za elektrotehniko, 2019. URL: <https://plus.cobiss.net/cobiss/si/sl/bib/13747203> (pridobljeno 25. 3. 2025)

Omenjena predavanja in vaje v tem priročniku se izvajajo pri obveznem strokovnem predmetu Robotski vid, na drugostopenjskem študiju robotike na Fakulteti za elektrotehniko Univerze v Ljubljani [25]. Za nadaljnje branje so pri vsaki temi v drsnicah predavanj navedeni sklici na ustrezno poglavje v eni izmed navedenih knjig:

- [11] Richard Szeliski. *Computer Vision: Algorithms and Applications, 2nd ed.* Predogled knjige za osebno rabo je dostopen tu: <https://szeliski.org/Book/>. Springer, 2022. URL: <https://link.springer.com/book/10.1007/978-3-030-34372-9>
- [6] Boštjan Likar. *Biomedicinska slikovna informatika in diagnostika*. Založba FE in FRI, 2008
- [5] Reinhard Klette. *Concise Computer Vision*. Springer, 2014
- [1] Wilhelm Burger in Mark J. Burge. *Principles of Digital Image Processing: Fundamental/ Core Techniques*. Springer, 2009

Zadnji del III je Dodatek, ki podaja osnovna matematična orodja kot so singularni razcep, toga aproksimacijska poravnava v 3D, ter uvod v uporabo programskih orodij kot so razvojna okolja Google Colab, Visual Studio Code, Conda in Docker ter Git za vodenje in sledenje razvoju programske kode. Za uvod v programski jezik Python in pregled osnovnih knjižnic priporočamo učbenik:

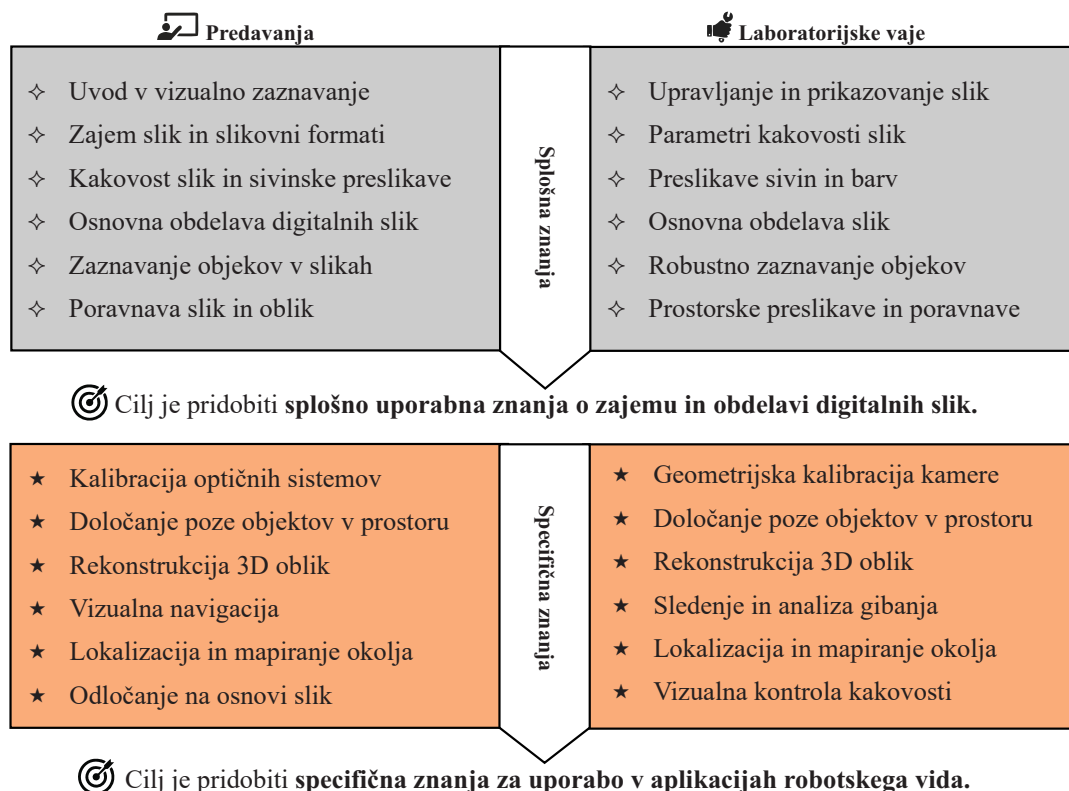
- [2] Miran Bürmen. *Uvod v programski jezik Python*. 1. izd. Ljubljana: Založba FE, 2016. ISBN: 978-961-243-318-5. URL: <https://plus.cobiss.net/cobiss/si/sl/bib/287674112> (pridobljeno 7. 4. 2025)

1.2.1 Minimalne strojne zahteve

Vaje so pripravljene tako, da se lahko izvajajo na prostodostopni oblačni računalniški platformi Google Colab (<https://colab.google>), zato bo za izvedbo vaj bo primeren katerikoli prenosnik z dostopom do spleta.

Za izvajanje kode na lastnem računalniku, ki vključuje v programskem jeziku Python napisane skripte z uporabo knjižnic OpenCV, TensorFlow in PyTorch za osnovne algoritme računalniškega vida ter učenje in testiranje globokih nevronske mreže na barvnih 2D slikah (npr. ImageNet [17]), je priporočljiva naslednja minimalna strojna oprema:

- **Procesor (CPU):** Intel Core i5 (10. generacija ali novejši) ali AMD Ryzen 5 (3000 serija ali novejši); vsaj 4 jedra / 8 niti (več je bolje za predprocesiranje podatkov).
- **Grafična kartica (GPU) [opcijsko]:** NVIDIA RTX 3060 (6GB VRAM) ali ekvivalentna kartica, alternativno AMD GPU s podporo ROCm (ni popolnoma združljivo s TensorFlow/PyTorch). Za poganjanje prednaučenih globokih modelov GPU ni nujno potreben, a bo v takem primeru izvajanje na CPU počasnejše
- **Pomnilnik (RAM):** 16GB DDR4, za večje nabor podatkov 32GB DDR4/DDR5.
- **Shranjevanje podatkov (SSD/HDD):** 512GB NVMe SSD, za nabor podatkov, kot je ImageNet večja kapaciteta, na primer 1TB NVMe SSD.
- **Operacijski sistem (OS):** Linux Ubuntu 24.04+ (za najboljšo združljivost z orodji za globoko učenje), Windows 10/11 (z uporabo WSL2 za boljšo združljivost s Python knjižnicami).
- **Dodatne zahteve:** Podpora za NVidia CUDA in cuDNN programski knjižnici (za GPU pospeševanje TensorFlow/PyTorch), nameščen Python 3.8+, dober hladilni sistem saj učenje



Slika 1.3: Primer učnega načrta z uporabo priročnika.

globokih modelov lahko pregrevanje.

Zakaj sta študij in delo na področju robotskega vida zanimiva?

Področje robotskega vida je eno najhitreje rastočih tehnoloških področij, saj združuje računalniški vid, umetno inteligenco, obdelavo signalov, robotiko in optimizacijo. Študij in delo na tem področju ponujata številne zanimive izzive, hkrati pa omogočata široke zaposlitvene možnosti in pomemben osebnostni razvoj.

Študentje, ki se ukvarjajo s tem področjem, se srečujejo z vrhunskimi tehnologijami, kot so globoke nevronske mreže, 3D rekonstrukcija, avtonomna navigacija in inteligentni vizualni sistemi. Študij robotskega vida ne razvija le tehničnih veščin (programiranje, obdelava podatkov, strojno učenje), temveč tudi analitično razmišljanje, sposobnost reševanja problemov in interdisciplinarno povezovanje znanja. Prav tako krepi ustvarjalnost, saj zahteva iskanje novih pristopov k zaznavanju in interpretaciji vizualnih informacij. Vsak nov napredek vodi v inovacije, ki imajo realne aplikacije v industriji in vsakdanjem življenju.

Povpraševanje po strokovnjakih za robotski vid in umetno inteligenco strmo narašča. Podjetja v avtomobilski industriji, robotiki, medicinski diagnostiki, avtomatizaciji proizvodnje, varnostnih sistemih in pametnih napravah iščejo kadre, ki obvladajo analizo slik in video signalov. Poleg klasičnih inženirskih vlog se odpirajo tudi priložnosti v raziskavah, razvoju in podjetništvu, v visokotehnoloških podjetjih doma in po vsem svetu, zato je robotski vid odlična izbira za karierno pot.

Obdelava digitalnih slik

2	Upravljanje in prikazovanje slik	25
2.1	Python knjižnica <code>numpy</code>	
2.2	Python knjižnica <code>pillow</code>	
2.3	Python knjižnica <code>matplotlib</code>	
2.4	Vaje z rešitvami	
2.5	Naloge in vprašanja	
3	Parametri kakovosti slik	35
3.1	Referenčni objekti in parametri kakovosti	
3.2	Histogram slike	
3.3	Vaje z rešitvami	
3.4	Naloge in vprašanja	
4	Preslikave svin in barv	49
4.1	Sivinske preslikave	
4.2	Preslikave med barvnimi prostori	
4.3	Vaje z rešitvami	
4.4	Naloge in vprašanja	
5	Osnovna obdelava slik	61
5.1	Filtriranje slik z diskretno konvolucijo	
5.2	Interpolacija slik	
5.3	Decimacija slik	
5.4	Vaje z rešitvami	
5.5	Naloge in vprašanja	
6	Robustno iskanje objektov	73
6.1	Gradient sivinske slike	
6.2	Harrisov detektor oglišč	
6.3	Cannyev detektor robov	
6.4	Houghov algoritem	
6.5	Konvolucijske nevronske mreže	
6.6	Vaje z rešitvami	
6.7	Naloge in vprašanja	
7	Prostorske preslikave in poravnave	97
7.1	Geometrijske preslikave	
7.2	Poravnava pripadajočih parov točk	
7.3	Vaje z rešitvami	
7.4	Naloge in vprašanja	

2. Upravljanje in prikazovanje slik

Vaja služi spoznavanju osnovnih ukazov za nalaganje, prikazovanje, shranjevanje slik in upravljanje z digitalnimi slikami v programskem jeziku Python. Spoznali boste nizkonivojske knjižnice za branje in pisanje poljubnih podatkovnih datotek, kot tudi knjižnice za branje in pisanje standardnih slikovnih formatov kot so .jpg, .png, .tif, itd.

2.1 Python knjižnica numpy

V jeziku Python lahko sivinske slike predstavimo z dvorazsežnimi polji `ndarray` v knjižnici `numpy`, v katerih so slikovni elementi običajno shranjeni kot nepreznadena 8-, 16- ali 32-bitna cela števila ali v zapisu s plavajočo vejico (Tabela 2.1).

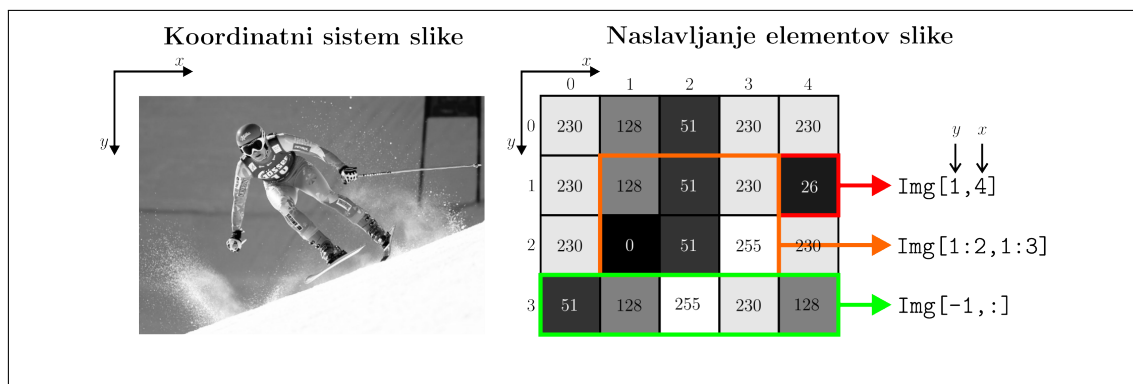
Tabela 2.1: Zapis sivinskih vrednosti s podatkovnimi tipi knjižnice `numpy`.

Zapis sivin	Podatkovni tip (dtype)	Zaloga vrednosti
binarna slika	'bool'	{ False, True }
8-bitni nepredznačeni	'uint8'	[0, 255]
16-bitni nepredznačeni	'uint16'	[0, 65535]
32-bitni predznačeni	'int32'	$[-2^{31}, 2^{31}-1]$
32-bitni s plavajočo vejico	'float32' ali 'single'	[0.0, 1.0]
64-bitni s plavajočo vejico	'float64' ali 'double'	[0.0, 1.0]

Knjižnico `numpy` naložimo z ukazom `import numpy as np`, do funkcij in spremenljivk v knjižnici pa dostopamo z `np. ....`. Nekatere uporabne funkcije za inicializacijo polja `ndarray` so `zeros()`, `ones()`, `zeros_like()`, `ones_like()`, `asarray()`, za pretvorbo tipa podatka `astype()` ali `array( mArray, dtype= ... )`, za branje števila dimenzij `ndim()` in velikosti `shape()` polja in za preoblikovanje polja `reshape()` in `transpose()`. Dvodimenzionalno polje `mArray` naslavljamemo z npr. `mArray[0,3]` (element v prvi vrstici, četrtem stolpcu), `mArray[:,1]` (drugi stolpec), `mArray[-1,:]` (zadnja vrstica), itd. Indekse v sliki za logične izraze nad elementi slike lahko iščemo z ukazom `where()`. Koordinatni sistem slike in primeri naslavljanja elementov v sliki so

prikazani na sliki 2.1.

Za branje in pisanje slik v surovem (nezgoščenem) zapisu sta v knjižnici numpy uporabni funkciji `fromfile()` in `tofile()`.



Slika 2.1: Koordinatni sistem digitalne slike in naslavljanje elementov slike.

2.2 Python knjižnica pillow

Za branje in pisanje slik v standardnih formatih (bmp, png, gif, eps, jpeg, itd.) pa lahko uporabimo knjižnico PIL in modul `PIL . Image` (če pri uvažanju knjižnice Python javi napako, potem jo najprej naložite v ukaznem oknu z ukazom `pip install pillow`). Knjižnica `pillow` (uvozi se kot modul `PIL`) je ena najbolj priljubljenih knjižnic za obdelavo slik v Pythonu in omogoča številne operacije, kot so nalaganje in shranjevanje slik, spreminjanje velikosti, obrezovanje, uporaba filtrov in druge transformacije slik. S `PIL` modulom sliko naložimo s funkcijo `open()`, ki ustvari spremenljivko tipa `Image`. Zapis slikovnih elementov v sliki preverimo z ukazom `getbands()`. S funkcijo `numpy . array` to spremenljivko pretvorimo v numpy podatkovno polje. Sliko v obliki numpy polja pretvorimo nazaj v tip `Image` s funkcijo `Image . fromarray()`, pretvorimo v poljuben format s funkcijo `Image . convert()` in shranimo z ukazom `save()`.

2.3 Python knjižnica matplotlib

Matplotlib je vodilna Python knjižnica za vizualizacijo podatkov, ki omogoča profesionalno prikazovanje grafov in slik. Osnovni moduli knjižnice so:

- `pyplot` - visokonivojski vmesnik za hitro ustvarjanje vizualizacij
- `Figure` - osnovni objekt, ki predstavlja celotno sliko
- `Axes` - posamezni grafi znotraj figure
- `Axis` - številčne osi grafa

Funkcije za prikazovanje slik so na voljo v modulu `matplotlib.pyplot`. Za izris slike je uporabna funkcija `imshow()`, za novo prikazno okno `figure()`, za urejanje osi pa `suptitle()`, `xlabel()` in `ylabel()`, `axes()`. Prikaz osvežite s funkcijo `show()`.

Matplotlib ponuja številne dodatne možnosti, vključno s 3D grafiki, animacijami in interaktivnimi prikazi. Za napredno uporabo je priporočljivo razumeti objektno-orientiran pristop prek `Figure` in `Axes` razredov.

2.4 Vaje z rešitvami

Koda vaje je dostopna na [Git repozitoriju](#).

Vaja 2.1 Z uporabo knjižnice PIL naložite sliko v datoteki `slika.jpg`, jo pretvorite v sivinsko sliko in shranite v formatu `png`. ■

Začnimo z uvozom potrebnih knjižnic kot je knjižnica `PIL.Image` (iz paketa `pillow`) omogoča delo s slikami. Prvi korak je nalaganje slike z ukazom `im.open('slika.jpg')`, ki prebere slikovno datoteko iz trenutnega imenika. Metoda `getbands()` nato vrne seznam barvnih kanalov, kar nam omogoča vpogled v strukturo slike (npr. RGB kanali za barvno sliko). Ta informacija je koristna za razumevanje, kako je slika sestavljena.

Glavna operacija je pretvorba barvne slike v sivinsko z metodo `convert('L')`. Ta pretvorba uporablja standardno formulo za izračun svetlosti (ang. *Luminance*), kjer vsak barvni kanal prispeva s specifično utežjo, na primer: rdeči kanal (R) prispeva 29.9%, zeleni (G) 58.7% in modri (B) 11.4%. Navedeni deleži oz. razmerje med njimi ustreza človeški percepciji svetlosti, saj je naše oko najbolj občutljivo na zeleno svetlobo.

Končno, sivinska slika se shrani v novo datoteko `slika.png` z metodo `save()`. Format datoteke se samodejno določi glede na podano končnico. V našem primeru uporabimo format PNG (ang. *Portable Network Graphics*) s končnico `.png`.

Rešitev lahko napišemo:

```
1 import PIL.Image as im
2 import numpy as np
3
4 pilImage = im.open('slika.jpg') # naloži sliko
5 print(pilImage.getbands()) # izpiši kanale naložene slike
6
7 # izvedi pretvorbo iz barvne v sivinsko sliko
8 # L = R * 299/1000 + G * 587/1000 + B * 114/1000
9 pilImageGray = pilImage.convert('L')
10
11 pilImageGray.save('slika.png') # shrani sliko
```

Koda izpiše rezultat v ukazni vrstici:

```
('R', 'G', 'B')
```

Alternativno lahko svetlost preračunamo tudi sami, po poljubni formuli. Uporabili bomo formulo za pretvorbo $L(u,v) = 0.299 \cdot R(u,v) + 0.587 \cdot G(u,v) + 0.114 \cdot B(u,v)$, $\forall(u,v)$, pri čemer uporabimo knjižnico `numpy`, ki ponuja podporo za numerične operacije. Rešitev lahko napišemo:

```
1 import PIL.Image as im
2 import numpy as np
3
4 pilImage = im.open('slika.jpg')
5 # pretvorba v numpy polje z decimalno vejico, ker bomo
6 # množili vrednosti s poljubnimi skalarji
7 arrImage = np.array(pilImage, dtype='float')
8
9 # funkcija za pretvorbo iz RGB v svetlost
10 def rgb2gray(rgb):
11     # varianta 1: razčlenjeno množenje z utežmi po RGB kanalih
12     # return 0.299 * rgb[...,0] + 0.587 * rgb[...,1] + 0.114 * rgb[...,2]
13     # varianta 2: razčlenjeno množenje z utežmi po RGB kanalih
14     return np.dot(rgb, [0.299, 0.587, 0.114]) # zapis s skalarnim produktom
15
16 # klic funkcije za dejansko izvedbo pretvorbe
```

```

17 arrImage = rgb2gray(arrImage)
18 # pretvorba v uint8 zapis pred shranjevanjem
19 pilImageGray = im.fromarray(arrImage.astype('uint8'))
20
21 pilImageGray.save( 'slika2.png' ) # shrani sliko

```

Vaja 2.2 Datoteka `slika-8bit.raw` vsebuje dvodimenzionalno (2D) sliko v obliki surovih oz. nezgoščenih podatkov (raw). Širina in višina slike sta $X \times Y = 975 \times 650$ slikovnih elementov, vsak slikovni element pa je zapisan kot nepredznačeno 8-bitno celo število. Napišite funkcijo za nalaganje poljubnih nezgoščenih slik:

```

1 def loadImageRaw( iPath, iSize, iFormat ):
2     # Tu napišite python kodo funkcije
3     return oImage

```

kjer je `iPath` pot do slike (mapa in ime datoteke), `iSize` vektor velikosti slike v slikovnih elementih, `iFormat` pa oblika zapisa oz. tip podatka. Funkcija naj vrne naloženo sliko `oImage` v obliki numpy polja. Naložite knjižnico `numpy` in uporabite funkcijo `fromfile()`. ■

Najprej uporabimo funkcijo `np.fromfile` za branje binarne datoteke v enodimenzionalno numpy polje, nato pa z `np.reshape` podatke reorganiziramo v dvodimenzionalno matriko (sliko) želene velikosti. Funkcija vrne sliko kot numpy polje, kar omogoča nadaljnjo obdelavo s knjižnicami za analizo slik. Rešitev lahko napišemo:

```

1 def loadImageRaw(iPath, iSize, iFormat):
2     '''
3     Naloži sliko iz raw datoteke
4
5     Parameters
6     -----
7     iPath : str
8         Pot do datoteke
9     iSize : tuple
10        Velikost slike
11     iFormat : str
12        Tip vhodnih podatkov
13
14     Returns
15     -----
16     oImage : numpy array
17        Izhodna slika
18
19     '''
20
21     oImage = np.fromfile(iPath, dtype=iFormat) # nalozi raw datoteko
22     oImage = np.reshape(oImage, iSize) # uredi v matriko
23
24     return oImage
25
26
27 # klic funkcije in izpis dimenzij numpy polja
28 arrImage = loadImageRaw('slika-8bit.raw', (650, 975), 'uint8')

```

```
29 | print(arrImage.shape)
```

Koda izpiše rezultat v ukazni vrstici:

```
(650, 975)
```

V gornjem primeru smo dodali še dokumentacijo funkcije, z opisom funkcionalnosti funkcije, opisom pomena in tipa vhodnih ter izhodnih spremenljivk. Tipična uporaba te funkcije vključuje nalaganje surovih slikovnih podatkov iz znanstvenih instrumentov ali medicinskih slikovnih sistemov, kjer se slike shranjujejo v neobdelani binarni obliki.

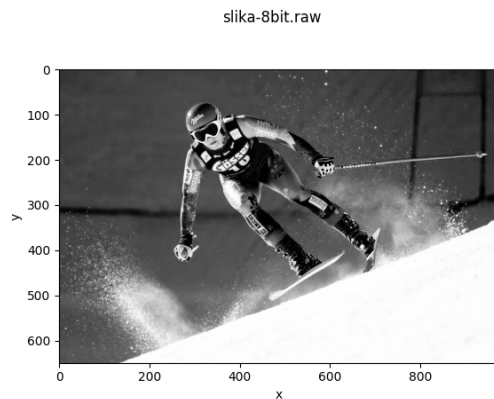
Vaja 2.3 Prikažite sliko na zaslon z uporabo funkcije `imshow()`. Prikaz sivin prilagodite zapisu s pomočjo parametra `cmap` (uporabite `cm.Greys_r` v knjižnici `matplotlib.cm`). Razmerje osi prikaza prilagodite razmerju osi slike z ukazom `matplotlib.axes().set_aspect('equal', 'datalim')`. Napišite funkcijo za prikazovanje poljubnih slik v obliki numpy polja:

```
1 | def showImage( iImage, iTitle ):
2 |     # Tu napišite python kodo funkcije
```

kjer je `iImage` slika, ki jo želite prikazati, `iTitle` pa naslov prikaznega okna. ■

Knjižnico `matplotlib` za vizualizacij najprej uvozimo, in sicer podmodula `pyplot` in `cm`. Rešitev lahko napišemo:

```
1 | import matplotlib.pyplot as plt
2 | import matplotlib.cm as cm # uvozi barvne lestvice
3 |
4 | def showImage(iImage, iTitle=''):
5 |     '''
6 |     Prikaže sliko iImage in jo naslovi z iTitle
7 |
8 |     Parameters
9 |     -----
10 |    iImage : numpy.ndarray
11 |        Vhodna slika
12 |    iTitle : str
13 |        Naslov za sliko
14 |
15 |    Returns
16 |    -----
17 |    Nothing
18 |
19 |    '''
20 |
21 |    plt.figure() # odpri novo prikazno okno
22 |    plt.imshow(iImage, cmap = cm.Greys_r) # prikaži sliko v novem oknu
23 |    plt.suptitle(iTitle) # označi naslov slike
24 |    plt.xlabel('x') # označi horizontalno os
25 |    plt.ylabel('y') # označi vertikalno os
26 |    plt.show() # prikaži sliko
27 |
28 | # primer uporabe
29 | showImage(arrImage, 'slika-8bit.raw')
```



Slika 2.2: Rezultat uporabe funkcije z ukazom `showImage(arrImage, 'slika-8bit.raw')`.

Funkcija najprej ustvari novo prikazno okno, nato uporabi `imshow` za prikaz slike z ustreznimi nastavitvami osi in naslova. Ključna prednost je možnost hitrega pregledovanja vmesnih rezultatov pri obdelavi slik, kar je še posebej uporabno med razvojem algoritmov. Funkcija ne vrača nobene vrednosti, saj je njena naloga zgolj vizualizacija. Slika 2.2 prikazuje dobljeni rezultat.

Vaja 2.4 Napišite funkcijo za shranjevanje slik v obliki numpy polja:

```
1 def saveImageRaw( iImage, iPath, iFormat ):
2     # Tu napišite python kodo funkcije
```

kjer je `iImage` slika, ki jo želite shraniti, `iPath` pot do slike (mapa in ime datoteke), `iFormat` pa oblika zapisa. Naložite knjižnico `numpy` in uporabite funkcijo `tofile()`. ■

Funkcija naj najprej pretvori podatke v določeni format z `astype()`, nato pa jih z `tofile()` zapiše v datoteko s funkcijo `tofile()`. Rešitev lahko napišemo:

```
1 def saveImageRaw(iImage, iPath, iFormat):
2     '''
3     Shrani sliko na disk
4
5     Parameters
6     -----
7     iImage : numpy.ndarray
8         Vhodna slika za shranjevanje
9     iPath : str
10        Pot in ime datoteke, v katero želimo sliko shraniti
11     iFormat : str
12        Tip podatkov v matriki slike
13
14     Returns
15     -----
16     Nothing
17     '''
18
19     iImage = iImage.astype(iFormat) # spremeni tip zapisa
20     iImage.tofile(iPath) # zapiši v datoteko
21
22 # primer uporabe
```



```
23 | saveImageRaw(arrImage, 'slika-8bit-2.raw', 'uint8')
```

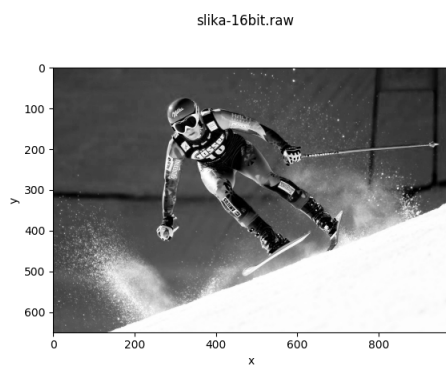
Ključne značilnosti funkcije so: (1) ohranjanje originalnih podatkov brez stiskanja, (2) možnost shranjevanja poljubnih numeričnih formatov in (3) enostavno branje s funkcijo `np.fromfile`. Funkcija ne vrača vrednosti, saj je njena naloga zgolj shranjevanje podatkov.

Vaja 2.5 Datoteka `slika-16bit.raw` vsebuje 2D sliko v nezgoščenem zapisu, slika velikosti $X \times Y = 975 \times 650$, vsak slikovni element pa je zapisan kot nepredznačeno 16-bitno celo število. Prilagodite in preizkusite delovanje zgoraj navedenih funkcij `loadImageRaw()`, `showImage()` in `saveImageRaw()`.

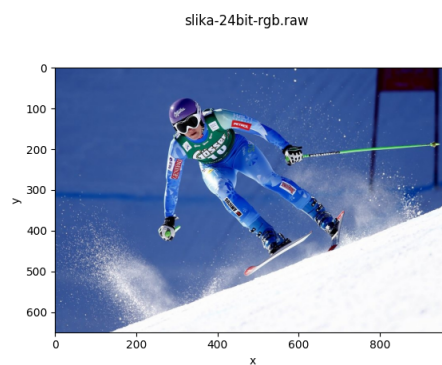
Za klic funkcij `loadImageRaw()` in `saveImageRaw()` določimo ustrezne argumente, kritičen argument je `iFormat`, ki ga moramo nastaviti na `'uint16'`. Funkcija `showImage()` je neodvisna od tipa zapisa, saj `imshow()` funkcija samodejno prilagodi prikaz. Rešitev lahko napišemo:

```
1 | arrImage = loadImageRaw('slika-16bit.raw', (650, 975), 'uint16')
2 | showImage(arrImage, 'slika-16bit.raw')
3 | saveImageRaw(arrImage, 'slika-16bit-2.raw', 'uint16')
```

Slika 2.3a prikazuje dobljeni rezultat.



(a)



(b)

Slika 2.3: Rezultat uporabe funkcije z ukazom (a) `showImage(arrImage, 'slika-16bit.raw')` v nalogi 2.5 in ukazom (b) `showImage(arrImage, 'slika-24bit-rgb.raw')` v nalogi 2.6.

Vaja 2.6 Datoteka `slika-24bit-rgb.raw` vsebuje barvno 2D sliko v nezgoščenem zapisu, slika ima velikost $X \times Y = 975 \times 650$, posamezen slikovni element pa je predstavljen s $24 = 3 \times 8$ biti. Barvna RGB slika je zapisana kot zaporedje treh nepreznanih 8 bitnih slik, kjer si sledijo rdeča (R), zelena (G) in modra (B) komponenta. Dopolnite funkcije `loadImageRaw()`, `showImage()` in `saveImageRaw()`, tako da bodo omogočale nalaganje, prikaz in shranjevanje barvnih RGB slik.

Za klic funkcij `loadImageRaw()` in `saveImageRaw()` določimo ustrezne argumente, kritičen argument je `iSize`, ki predstavlja dimenzije slike v prvih dveh vrednostih in število barvnih kanalov v tretji dimenziji (3 v primeru RGB barvne slike). Funkcija `showImage()` je neodvisna od tipa zapisa, saj `imshow()` funkcija samodejno prilagodi prikaz. Rešitev lahko napišemo:

```
1 | arrImage = loadImageRaw('slika-24bit-rgb.raw', (650, 975, 3), 'uint8')
2 | showImage(arrImage, 'slika-24bit-rgb.raw')
```

```
3 | saveImageRaw(arrImage, 'slika-24bit-rgb-2.raw', 'uint8')
```

Slika 2.3b prikazuje dobljeni rezultat.

Opomba: Vse dosedaj napisane funkcije bomo shranili v datoteko `rvlib.py` za uporabo pri naslednjih vajah.

2.5 Naloge in vprašanja

Poročilo naloge pripravite v obliki Python skripte, ki naj izvede zahtevane izračune oz. klice funkcij in izriše pripadajoče slike oz. grafe. Vaša dolžnost je, da v skripti uvozite ustrezne Python knjižnice in dodate morebitne podporne skripte. Skripta se mora izvesti in poustvariti rezultate, kot se zahtevajo v nalogah. Na vprašanja odgovorite v obliki komentarja, na primer *# Naloga X.2: To je moj odgovor..*

Naloga 2.1 Napišite funkcijo za nalaganje slik v standardnih formatih (bmp, jpg, png, tif, gif, idr.) s knjižnico `PIL.Image`:

```
1 | def loadImage( iPath ):
2 |     # Tu napišite python kodo funkcije
3 |     return oImage
```

kjer je `iPath` pot do slike (mapa in ime datoteke). Funkcija naj vrne sliko v obliki `numpy.ndarray` polja v spremenljivki `oImage`. Naložite in prikažite sliko `slika.jpg`.

Naloga 2.2 Napišite funkcijo za shranjevanje slik v standardnih formatih (bmp, jpg, png, tif, gif, idr.) s knjižnico `PIL.Image`:

```
1 | def saveImage( iPath, iImage, iFormat ):
2 |     # Tu napišite python kodo funkcije
```

kjer je `iPath` pot do slike (mapa in ime datoteke), spremenljivka `iImage` slika v obliki `numpy.ndarray` polja, `iFormat` pa beseda z željeno obliko zapisa slike (npr. `'bmp'`, `'png'`).

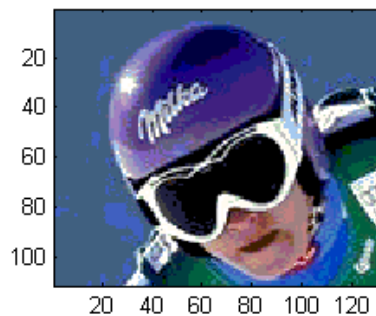
Naloga 2.3 Iz barvne 2D slike `slika.jpg` izluščite pravokotno barvno podsliko med oglišči $(x_1, y_1) = (260, 70)$ in $(x_2, y_2) = (390, 180)$ in jo prikažite. Preverite pravilnost določanja področja s pomočjo slike 2.4.

Naloga 2.4 S pomočjo indeksiranja prezrcalite sliko `slika.jpg` preko navpične osi in jo prikažite.

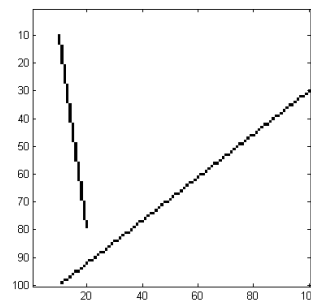
Naloga 2.5 Uporabite funkcijo `numpy.transpose` za rotacijo slike za 90 stopinj, tako da transponirate `x` in `y` os slike `slika.jpg`.

Naloga 2.6 Pretvorite barvno sliko $C = [R, G, B]$ v sivinsko sliko S preko enačbe $S = \frac{1}{3}R + \frac{1}{3}G + \frac{1}{3}B$ in narišite projekciji maksimalne in povprečne svetlosti v smeri x in y koordinat 2D slik. Projekciji lahko ustvarite z uporabo `for` zanke ali funkcij `numpy.max()` in `numpy.mean()`, pri čemer ustrezno nastavite parameter `axis`. Za izris projekcij uporabite funkcijo `plot()` v knjižnici `matplotlib.pyplot`.

Naloga 2.7 Zamenjajte vrednosti slikovnih elementov rdečega kanala 2D barvne slike, ki imajo vrednost med 160 in 200 z vrednostjo 255. Uporabite funkcijo `where()` v knjižnici `numpy`. Prikažite originalno in novo barvno sliko.



Slika 2.4: Izrez barvne slike na pravokotnem področju med oglišči $(x_1, y_1) = (260, 70)$ in $(x_2, y_2) = (390, 180)$.



Slika 2.5: Izris daljic s sivinskimi vrednostmi 0 in krajiščema $(x_1, y_1) = (10, 10)$ in $(x_2, y_2) = (20, 80)$ ter $(x_1, y_1) = (100, 30)$ in $(x_2, y_2) = (10, 100)$.

Naloga 2.8 S pomočjo Bresenhamovega algoritma 2.1 napišite funkcijo za risanje digitalnih daljic v sliko:

```
1 def drawLine( iImage, iValue, x1, y1, x2, y2 ):
2     # Tu napišite python kodo funkcije
3     return oImage
```

kjer je `iImage` slika, v katero želite risati daljico, `iValue` vrednost, ki jo pripišete slikovnemu elementu na daljici, (x_1, y_1) in (x_2, y_2) pa so koordinate začetne in končne točke daljice. Preverite delovanje funkcije, tako da ustvarite belo sivinsko sliko velikosti $X \times Y = 100 \times 100$, nato vrišite dve črni daljici s krajiščema $(x_1, y_1) = (10, 10)$ in $(x_2, y_2) = (20, 80)$ ter $(x_1, y_1) = (100, 30)$ in $(x_2, y_2) = (10, 100)$. Dobljeno sliko primerjajte s sliko 2.5.

Podrobna razlaga delovanja algoritma je dana v [22].

Algoritem 2.1 — Bresenhamov algoritem.

```
funkcija narišiDaljico(slika, vrednost, x1, y1, x2, y2)
    dx := abs(x2-x1)
    dy := abs(y2-y1)
    če x1 < x2 potem sx := 1 sicer sx := -1
    če y1 < y2 potem sy := 1 sicer sy := -1
    napaka := dx-dy
zanka
    slika(x1,y1) = vrednost
    če x1 = x2 in y1 = y2 potem
        izhod iz zanke
    e2 := 2*napaka
    če e2 > -dy potem
        napaka := napaka - dy
        x1 := x1 + sx
    če e2 < dx potem
        napaka := napaka + dx
        y1 := y1 + sy
konec zanke
```

Naloga 2.9 Spremenite funkcijo za risanje daljic `drawLine()` tako, da bo omogočala risanje

daljic v RGB barvno sliko z izbrano barvo. Ustvarite 2D barvno sliko velikosti $X \times Y = 300 \times 100$ in vse slikovne elemente obarvajte belo. S pomočjo posodobljene funkcije za risanje barvnih daljic v 2D sliko izpišite vaše ime, pri čemer ročno določite ustrezna krajišča daljic. Vsaka črka vašega imena naj ima drugo barvo.



3. Parametri kakovosti slik

Kakovost zajete slike je odvisna od številnih dejavnikov v procesu zajemanja slik, ki predstavlja preslikavo iz fizičnega v slikovni prostor. Pomembni dejavniki, ki vplivajo na kakovost slik so osvetlitev, gostota vzorčenja, napake oz. aberacije optičnega sistema, idr.

Popačenje slik kot posledico zajema v splošnem imenujemo **šum**; to je izraz za kakršnokoli naključno spreminjanje sivinskih vrednosti in ima lahko pomemben vpliv na kakovost slike. Izvor, vrsta in velikost šuma so odvisni od fizikalnega procesa oziroma načina zajemanja slik, najbolj znani in splošno uporabni model šuma pa je Gaussov šum. Gaussov šum je definiran s funkcijo gostote verjetnosti $p(z) = (\sqrt{2\pi}\sigma)^{-1} \exp(-(z - \mu)^2/2\sigma^2)$, kjer μ predstavlja srednjo vrednost, σ pa standardno deviacijo šuma. Šum v sliki ocenjujemo na področjih s predvidoma konstantno sivinsko vrednostjo.

Popačenje slik, ki povzroča **sivinske nehomogenosti**, pogosto izvira iz različnih dejavnikov, kot so neenakomerna osvetlitev, šum senzorjev v kameri, optične aberacije ali nepravilnosti v procesiranju signala. Na primer, pri slikanju z digitalnimi napravami lahko šum, kot je termični ali fotonski šum, povzroči naključne spremembe v intenziteti svin na posameznih pikslih, medtem ko neenakomerna osvetlitev ustvari svetlostne razlike v vidnem polju. Za odpravljanje tega popačenja se uporabljajo metode, kot so korekcija osvetlitve z uporabo kalibracijskih slik (npr. "flat-field" korekcija, ki vključuje slikanje homogene temne in/ali svetle podlage), filtriranje šuma z algoritmi, kot je lokalno povprečenje ali Gaussov filter, ter naprednejše tehnike, kot je homomorfno filtriranje. V kompleksnejših primerih se lahko uporabi tudi strojno učenje za prepoznavanje in odstranjevanje specifičnih vzorcev popačenj, kar zagotovi bolj homogeno in kakovostno sliko.

Optične aberacije so napake v slikovnem sistemu, ki povzročajo odstopanja od idealnega projiciranja svetlobe na slikovni senzor, kar posledično vpliva na ostrino, kontrast in popolnost zajete slike. Aberacije nastanejo zaradi neidealnosti leč ali zrcal, ki povzročajo, da se svetlobni žarki iz enake točke predmeta ne srečajo povsem točno na slikovni ravnini. Aberacije delimo na **kromatske** (nastanejo zaradi različnega loma svetlobe različnih valovnih dolžin) in **monokromatske** (pojavi se tudi pri enobarvni svetlobi).

Med monokromatske aberacije spadajo:

- Sferična aberacija (neravnomerno lomljenje žarkov na robovih leče)

- Koma (razmazovanje točkastih virov svetlobe v obliki repa)
- Astigmatizem (različna goriščna razdalja v navpični in vodoravni smeri)
- Ukrivljenost polja (slika se fokusira na ukrivljeno površino namesto na ravnino)
- Geometrijska distorzija (deformacija slike, bodisi blazinasta ali sodčkasta)

Najpogostejša je **sferična aberacija**, ki zmanjšuje kontrast in ostrino slike, še posebej na robovih polja. *Prospektivno* jo lahko zmanjšamo z uporabo asferičnih leč, kombinacij več leč ali diafragmiranjem (zožanje odprtine, kar izključi skrajne žarke). *Retrospektivno* pa jo lahko delno popravimo z algoritmi za obdelavo slik, kot je dekonvolucija z znano funkcijo razširitve točke (PSF; ang. *Point Spread Function*) ali prilagajanjem kontrasta z digitalno obdelavo.

Z modeliranjem prostorskih popačitev lahko retrospektivno odpravimo **geometrijsko distorzijo**, ki jo bomo obravnavali v poglavju 8 o geometrijski kalibraciji kamere, ki je nujna za izvajanja točnih geometrijskih merjenj objektov zanimanja na podlagi zajetih slik.

Kljub ukrepom za zmanjšanje aberacij in drugih popačenj je njihova popolna odprava teoretično nemogoča, zato je treba pri načrtovanju optičnih sistemov poiskati optimalno ravnovesje med kompleksnostjo, ceno in kakovostjo slike.

3.1 Referenčni objekti in parametri kakovosti

Kakovost slik lahko ovrednotimo z zajemom slik znanih kalibracijskih objektov (Slika 3.1), iz katerih izračunamo parametre kakovosti slik kot so kontrast, prostorska in sivinska ločljivost ter šum oz. razmerje signal–šum (Slika 3.2).

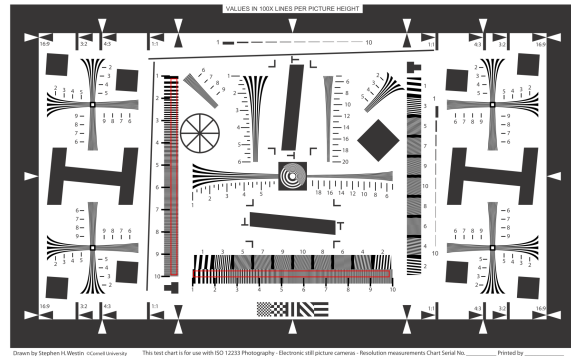
Ocenjevanje parametrov kakovosti slik vključuje analizo več ključnih lastnosti, ki vplivajo na zaznavanje in uporabnost slike: kontrast, prostorska ločljivost, sivinska ločljivost ter šum oziroma razmerje signal–šum (SNR). **Kontrast** meri razliko v svetlosti med najsvetlejšimi in najtemnejšimi deli slike; višji kontrast omogoča boljše razločnost podrobnosti. Ocenjuje se ga lahko z uporabo histogramov ali metrik, kot je Michelsonov kontrast. **Prostorska ločljivost** opisuje sposobnost slike, da prikaže fine podrobnosti, in je odvisna od števila pikslov na dolžinsko enoto (npr. *dots per inch* oz. DPI); meri se z vzorci, kot je USAF 1951 testna tarča, ki vsebuje črte različnih gostot. **Sivinska ločljivost** se nanaša na število razločljivih sivinskih nivojev (npr. 8-bitna slika ima 256 nivojev) in vpliva na gladkost prehodov med odtenki; ocenjuje se z analizo bitne globine in vizualnimi testi. **Šum** označuje neželene naključne variacije v intenziteti pikslov, medtem ko **razmerje signal–šum (SNR)** primerja moč uporabnega signala z močjo šuma, izraženo v decibelih (dB) – višji SNR pomeni boljše kakovost. Za standardizirano ocenjevanje se uporabljajo kalibri, kot so že omenjena USAF 1951 tarča za prostorsko ločljivost, Siemensova zvezda za testiranje ostrine, ali ISO 12233 tarča, ki združuje meritve ločljivosti in kontrasta. Za šum in SNR se pogosto uporabljajo enobarvne testne slike ali specializirani vzorci, kot so tisti iz standarda ISO 15739. Ti kalibri omogočajo objektivno primerjavo in optimizacijo slikovnih sistemov.

3.2 Histogram slike

Histogram slike je grafično orodje za prikazovanje frekvenčnih porazdelitev sivinskih vrednosti slikovnih elementov na sliki (Slika 3.3a). Vrednosti k na abscisni osi histograma predstavljajo dinamično območje sivinskih vrednosti slike. Za izbrano sivinsko vrednost na abscisni osi predstavlja vrednost na ordinatni osi h_k število slikovnih elementov na sliki z izbrano sivinsko vrednostjo (Slika 3.3b). Normaliziran histogram predstavlja oceno relativne verjetnostne porazdelitve $p_r(z)$; v primeru karakterizacije šuma na področju slike s približno konstantno intenziteto lahko na podlagi $p_r(z)$ določimo parametra μ in σ Gaussove (oz. normalne) porazdelitve šuma kot $\mu = \sum_{i=1}^L z_i p_r(z_i)$ in $\sigma^2 = \sum_{i=1}^L (z_i - \mu)^2 p_r(z_i)$. Parameter σ je v tem primeru sorazmeren *magnitudi šuma* in odraža širino Gaussove porazdelitve.

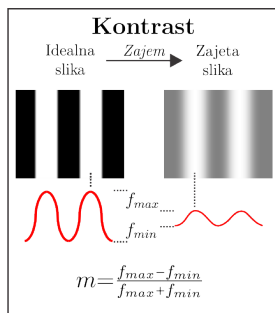


(a) ColorChecker

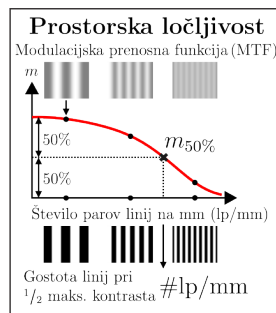


(b) ISO 12233

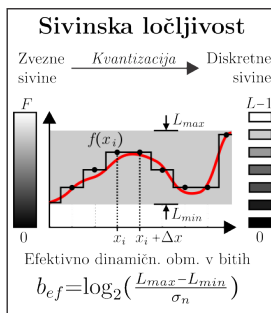
Slika 3.1: Sliki kalibracijskih objektov, ki ju uporabljamo pri vrednotenju (a) kakovosti oslikave sivin in barv, npr. z izračunom sivinske ločljivosti in razmerja signal–šum, in (b) prostorske ločljivosti z modulacijsko prenosno funkcijo.



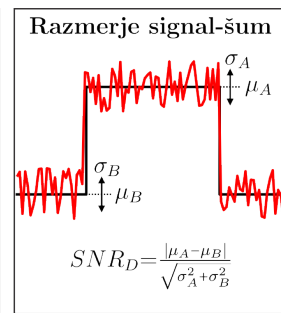
(a)



(b)

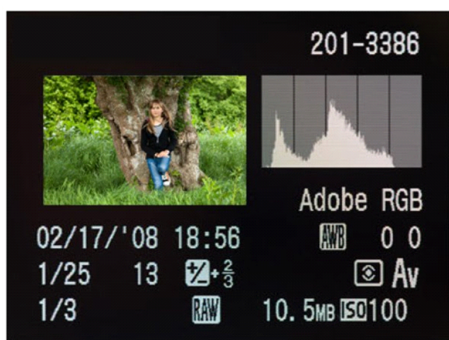


(c)

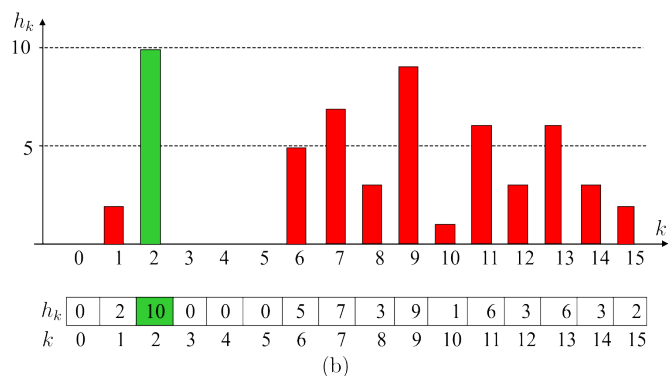


(d)

Slika 3.2: Parametri kakovosti slik: (a) Michelsonov kontrast na periodičnih vzorcih spremenljive frekvence, (b) prostorska ločljivost na podlagi poteka Michelsonovega kontrasta; (c) sivinska ločljivost, ki zavisi od dinamičnega območja (kvantizacije) in šuma; (d) razmerje signal–šum, ki se lahko uporabi za določanje razločljivosti objektov zanimanja.



(a)



(b)

Slika 3.3: (a) Barvna slika in histogram pripadajoče sivinske slike. (b) Histogram nepredznačene 4-bitne sivinske slike z dinamičnim območjem 0–16, številom predalov 16 in vrednostmi h_k .

3.3 Vaje z rešitvami

Vaja je namenjena spoznavanju in razumevanju določanja parametrov kakovosti slik ter preizkušanju vpliva zunanjih dejavnikov na kakovost zajete slike. Koda vaje je dostopna na [Git repozitoriju](#).

Vaja 3.1 S pomočjo telefona oz. fotoaparata zajemite dve sliki kalibra ColorChecker (Slika 3.1a), prvo ko so v predavalnici prižgane luči in drugo ko so luči ugasnjene. Bliskavica naj bo pri tem izklopljena. Ob prižganih lučeh zajemite še sliko kalibra ISO 12233 (Slika 3.1b). Slike prenesite na računalnik in jih naložite Python s funkcijo `loadImage()` (Vaja 2). ■

Potrebne Python knjižnice in vse dosedaj napisane funkcije bomo najprej naložili iz naše knjižnice `rvlib.py`, nato zajeti sliki naložimo in prikazemo:

```
1 import matplotlib.pyplot as plt
2 import PIL.Image as im
3 import numpy as np
4 from rvlib import *
5
6 # naloži sliki
7 col_svetla = loadImage('slika1_col_svetla.jpg')
8 col_temna = loadImage('slika2_col_temna.jpg')
9
10 # prikaži sliki
11 showImage(col_svetla, 'Osvetljen kaliber')
12 showImage(col_temna, 'Neosvetljen kaliber')
```

Primer zajetih slik prikazuje slika 3.4.



(a)



(b)

Slika 3.4: Zajeti sliki ob (a) prižganih in (b) ugasnjenih lučeh v predavalnici.

Vaja 3.2 Zajete slike pretvorite v sivinske slike po enačbi $S = 0,299R + 0,587G + 0,114B$. Za shranjevanje in kasnejše nalaganje spremenljivk v obliki polja `numpy.ndarray` iz oziroma v datoteko uporabite funkciji `saveImageRaw()` in `loadImageRaw()` (Vaja 2). ■

```
1 def colorToGray(iImage):
2     '''
3     Pretvori barvno sliko v sivinsko.
4
5     Parameters
6     -----
7     iImage : numpy.ndarray
8         Vhodna barvna slika
9
10    Returns
11    -----
12    oImage : numpy.ndarray
13        Sivinska slika
```

```

14     '''
15     dtype = iImage.dtype
16     r = iImage[0,:,:].astype('float')
17     g = iImage[1,:,:].astype('float')
18     b = iImage[2,:,:].astype('float')
19
20     return (r*0.299 + g*0.587 + b*0.114).astype(dtype)
21
22 # pretvori obe sliki v sivinski
23 gray_svetla = colorToGray(col_svetla)
24 gray_temna = colorToGray(col_temna)
25
26 # shrani kot surove podatke v .raw datoteki
27 saveImageRaw(gray_svetla, 'slika1_gray_svetla.raw', 'uint8')
28 saveImageRaw(gray_temna, 'slika2_gray_temna.raw', 'uint8')
29
30 # naloži surove podatke iz .raw datotek
31 gray_svetla = loadImageRaw('slika1_gray_svetla.raw', (1836, 3264), 'uint8')
32 gray_temna = loadImageRaw('slika2_gray_temna.raw', (1836, 3264), 'uint8')

```

Vaja 3.3 Napišite funkcijo za izračun in prikaz histograma sivinske slike:

```

1 def computeHistogram( iImage, iNumBins, iRange=[], iDisplay=False, iTitle=''
2 ):
3     # Tu napišite python kodo funkcije
4     return oHist, oEdges

```

kjer je `iImage` slika, za katero se histogram računa, `iNumBins` število predalov as histograma, `iRange` pa spremenljivka tipa `list`, ki podaja območje vrednosti histograma kot par minimalne in maksimalne vrednosti (npr. `[0, 255]` za nepredznačeno 8-bitno sliko). Histogram naj se prikaže le, kadar ima vhodna spremenljivka `iDisplay` vrednost `True` (privzeto `False`), `iTitle` pa naj bo naslov prikaznega okna, v katerem je histogram prikazan. Za prikazovanje pa uporabite funkcijo `bar()` v Python knjižnici `matplotlib.pyplot`.

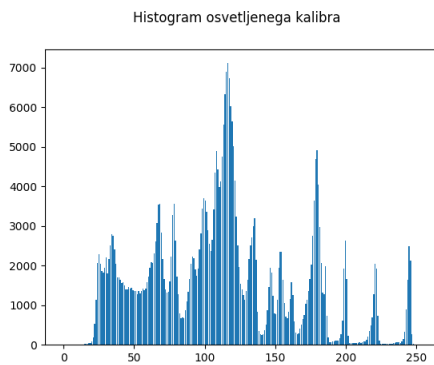
Funkcija vrne histogram slike `oHist` v obliki enodimenzionalnega polja tipa `numpy.array`, katerega dolžina je enaka številu predalov, kot jih podaja vhodna spremenljivka `iNumBins`. Izhodna spremenljivka `oEdges` naj vrne meje predalov histograma v obliki enodimenzionalnega polja tipa `numpy.array`, pri čemer je dolžina za ena večja od števila predalov. ■

Funkcijo `computeHistogram()` zastavimo tako, da izračuna histogram slike `iImage` z razvrščanjem vrednosti slikovnih pik v `iNumBins` enako širokih intervalov (binov) med najmanjšo (`iMin`) in največjo (`iMax`) vrednostjo, ki ju lahko podamo z `iRange` ali ju funkcija določi sama. Nato prešteje, koliko pik spada v vsak interval, in vrne histogram (`oHist`) ter meje intervalov (`oEdges`). Če je vhodni parameter `iDisplay=True`, prikaže histogram kot stolpični graf z naslovom `iTitle`. Funkcija uporablja NumPy za vektorske operacije in Matplotlib za vizualizacijo. Rešitev lahko napišemo:

```

1 def computeHistogram(iImage, iNumBins, iRange=[], iDisplay=False, iTitle=''):
2     iImage = np.asarray(iImage)
3     iRange = np.asarray(iRange)
4     if iRange.size == 2:
5         iMin, iMax = iRange
6     else:

```



Slika 3.5: Histogram slike 3.4a.

```

7     iMin, iMax = np.min(iImage), np.max(iImage)
8     oEdges = np.linspace(iMin, iMax+1, iNumBins+1)
9     oHist = np.zeros([iNumBins,])
10    for i in range(iNumBins):
11        idx = np.where((iImage >= oEdges[i]) * (iImage < oEdges[i+1]))
12        if idx[0].size > 0:
13            oHist[i] = idx[0].size
14    if iDisplay:
15        plt.figure()
16        plt.bar(oEdges[:-1], oHist)
17        plt.suptitle(iTitle)
18
19    return oHist, oEdges
20
21 # primer izračuna in prikaza histograma svetle slike
22 oHist, oEdges = computeHistogram(gray_svetla[:, :4], 255, [0, 255], \
23                                 True, 'Histogram osvetljenega kalibra')

```

Slika 3.5 prikazuje histogram slike 3.4a.

Vaja 3.4 Iz zajetih slik kalibra ColorChecker izluščite pravokotne podslike tako, da bo vsaka podslika vsebovala le eno področje 1–24. Podslike tipa `numpy.ndarray` shranite v vrstični vektor tipa `list`, v vrstnem redu kot je označen na sliki (1–24). Za hitro in enostavno določanje pravokotnih področij na sliki lahko označite le položaje nasprotnih si oglišč pravokotnika, tako da uporabite funkcijo `ginput()` v knjižnici `matplotlib.pyplot`. ■

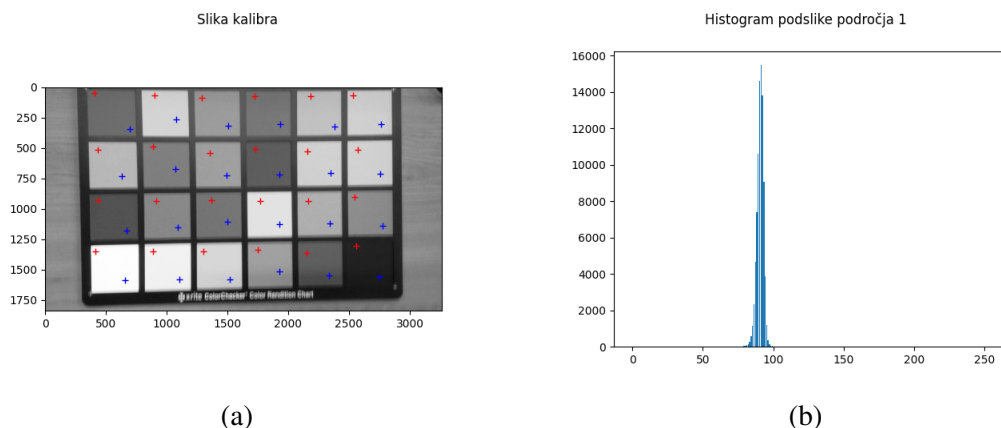
Naložimo sliko kalibra ColorChecker in ročno označimo oglišča podpodročij ter shranimo točke:

```

1 plt.close('all')
2 showImage(gray_svetla, 'slika kalibra ColorChecker')
3 plt.ioff() # izključi interaktivni način
4 pts = plt.ginput(n=48, timeout=-1)
5 pts = np.array(pts, dtype='float')
6 saveImageRaw(pts, 'nalog4_pts.raw', 'float')

```

Označene točke si lahko prikažemo na sivinski sliki kalibra in s histogramom vizualiziramo porazdelitev intenzitet in tako kvalitativno ocenimo stopnjo šuma:



Slika 3.6: Slika (a) prikazuje označene točke, slika (b) histogram gornjega levega področja z oznako 1.

```

1 pts = loadImageRaw('data/naloga4_pts.raw', (48, 2), 'float')
2 plt.close('all')
3
4 plt.imshow(gray_svetla, cmap = cm.Greys_r)
5 plt.suptitle('Slika kalibra')
6
7 # nariši označene točke
8 pts = pts.reshape((24,2,2))
9 sub_gray_svetla = []
10 for i in range(np.size(pts,axis=0)):
11     plt.plot(pts[i,0,0],pts[i,0,1],'+r')
12     plt.plot(pts[i,1,0],pts[i,1,1],'+b')
13     xs = round(pts[i,0,0])
14     ys = round(pts[i,0,1])
15     xe = round(pts[i,1,0])
16     ye = round(pts[i,1,1])
17     sub_gray_svetla.append(gray_svetla[ys:ye,xs:xe])
18 plt.show(block = True)
19
20 computeHistogram(sub_gray_svetla[0], 255, [0, 255], \
21                 True, 'Histogram podslike področja 1')
```

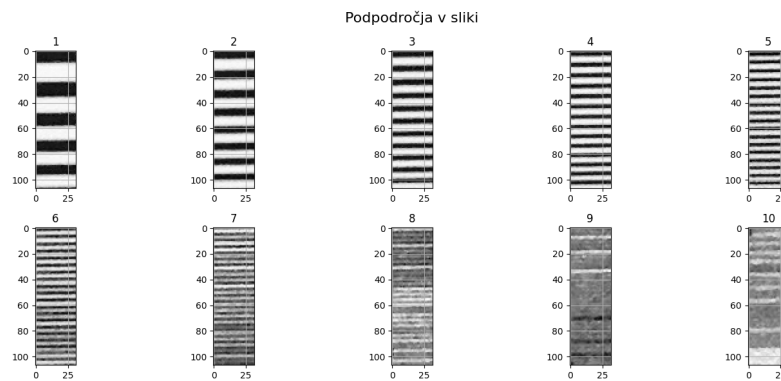
Rezultat izvajanja gornjega bloka Python kode je prikazan v sliki 3.6.

Vaja 3.5 Iz zajete slike kalibra ISO 12233 izluščite pravokotni podsliki vodoravnih in navpičnih linijskih parov tako, kot označujeta rdeča pravokotnika na sliki 3.1b. Vsako od podslik tipa `numpy.ndarray` razrežite na 10 podslik vzdolž daljše osi in jih shranite v vektor tipa `list`, v vrstnem redu od podslik z najredkejšimi do najgostejšimi linijami (1–10). ■

Zajeto sliko kalibra ISO 12233 prikažemo in ročno označimo oglišči pravokotnega podpodročja, kjer se nahajajo bodisi horizontalni bodisi vertikalni vzorci črno-belih linij. Vzorci linij naj bodo nujno znotraj pravokotnega okvirja, brez belega ozadja, za pravilen izračun Michelsonovega kontrasta. V primeru, da je slika zajeta *postrani* ponovite zajem tako, da bodo vzorci v smeri osi slike. Uporabimo lahko naslednjo kodo:

```

1 gray_space = colorToGray(loadImage('slika3_space_res.jpg'))
2 plt.close('all')
```



Slika 3.7: Slika prikazuje izluščena podpodročja vertikalnih črno-belih prog kalibra ISO12233.

```
3 showImage( gray_space, 'Slika kalibra ISO 12233' )
4 plt.ioff() # turn off interactive mode
5 pts = plt.ginput(n=2, timeout=-1)
6 pts = np.array(pts, dtype='float')
7 saveImageRaw(pts, 'naloga5_pts.raw', 'float')
```

Na osnovi oglišč v datoteki 'naloga5_pts.raw' izluščimo pravokotno sliko in jo razrežemo na 10 enakih delov. V spodnji kodi smo predvideli, da so bili označeni vertikalni vzorci, zato razrez naredimo po y osi. Rešitev lahko napišemo:

```
1 pts = loadImageRaw('naloga5_pts.raw', (2, 2), 'float')
2 plt.ion()
3
4 xs = np.round(pts[0,0])
5 xe = np.round(pts[1,0])
6 yc = np.round(np.linspace(pts[0,1],pts[1,1],11))
7 sub_gray_space = []
8 for i in range(len(yc)-1):
9     sub_gray_space.append(gray_space[yc[i]:yc[i+1],xs:xe])
10
11 # prikaz
12 fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(15, 6)) # matrika 2x5
13     sličic
14 fig.suptitle("Podpodročja v sliki", fontsize=16)
15 axes = axes.flatten() # indekse osi pretvorimo v vektor za lažji izris
16
17 # Plot data on each subplot
18 for i, ax in enumerate(axes):
19     ax.imshow(sub_gray_space[i], cmap = cm.Greys_r) # izris sličice
20     ax.set_title(f"{i+1}") # naslov sličice
21     ax.grid(True) # prikaz oslonilnih črt
22 plt.tight_layout()
23 plt.show(block=True)
```

Slika 3.7 prikazuje izluščena podpodročja v sliki kalibra ISO12233.

Vaja 3.6 Napišite funkcijo za izračun mere kontrasta m na sivinskih podslakah kalibra ISO 12233 iImages:

```
1 def computeContrast( iImages ):
2     # Tu napišite python kodo funkcije
3     return oM
```

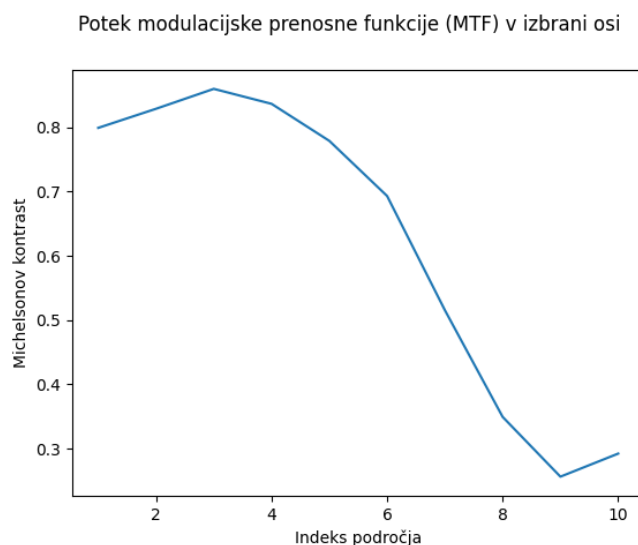
ki vrne polje skalarnih vrednosti oM, ki ima število elementov enako številu vhodnih slik. Minimalno in maksimalno vrednost f_{min} in f_{max} določite kot 5- in 95-percentil vseh sivinskih vrednosti tako, da uporabite funkcijo percentile() v knjižnici numpy. Preverite, da je kontrast najvišji pri podslakah z najredkejšimi pari vodoravnih oz. navpičnih linij.

Prikažite modulacijsko prenosno funkcijo (MTF; ang. *Modulation Transfer Function*) in določite prostorsko ločljivost sistema pri polovični vrednosti kontrasta glede na najvišjo vrednost.

```
1 def computeContrast(iImages):
2     '''
3     Izracunaj kontrast slik
4
5     Parameters
6     -----
7     iImages : list of numpy.ndarray
8         Vhodne slike, na katerih želimo izracunati kontrast
9
10    Returns : list
11        Seznam kontrastov za vsako vhodno sliko
12    '''
13    oM = np.zeros((len(iImages),))
14    for i in range(len(iImages)):
15        fmin = np.percentile(iImages[i].flatten(),5)
16        fmax = np.percentile(iImages[i].flatten(),95)
17        oM[i] = (fmax - fmin)/(fmax + fmin)
18    return oM
19
20 # prikaz modulacijske prenosne funkcije
21 mtf = computeContrast(sub_gray_space)
22 plt.figure()
23 plt.plot(mtf)
24 plt.suptitle('Potek modulacijske prenosne funkcije (MTF) v izbrani osi')
25 plt.xlabel('Indeks področja')
26 plt.ylabel('Michelsonov kontrast')
27 plot.show(block=True)
```

Izračun števila parov črt na palec (LPPI) za ISO 12233 grafikon naredimo glede na velikost kalibra, ki smo ga slikali. Vrednosti od 1 do 10 na ISO 12233 grafikonu predstavljajo 100-kratno število parov črt na višino slike (LP/PH), kjer je višina slike v našem primeru 180 mm (7,09 palca). To pomeni, da vzorec z oznako n ustreza:

$$LP/PH = n \times 100 \quad (3.1)$$



Slika 3.8: Slika prikazuje modulacijsko prenosno funkcijo.

Za izračun dejanskega števila parov črt na palec (LPPI; ang. *Line Pairs Per Inch*; 1 palec $\approx 25,4$ mm) uporabimo naslednje korake:

- Določimo dejansko višino grafikona v palcih:

$$H = 180 \text{ mm} = 7,09 \text{ palca} \quad (3.2)$$

- Ker gre za vertikalno ločljivost (horizontalne črte) izračunamo LPPI kot^a:

$$\text{LPPI} = \frac{\text{LP/PH}}{H} = \frac{n \times 100}{7,09} \approx n \times 14,11 \quad (3.3)$$

Tako dobimo naslednje vrednosti za posamezne vzorce:

Oznaka (n)	LP/PH	LPPI
1	100	14,1
2	200	28,2
3	300	42,3
4	400	56,4
5	500	70,6
6	600	84,7
7	700	98,8
8	800	112,9
9	900	127,0
10	1000	141,1

^aV primeru, da gre za horizontalno ločljivost (vertikalne črte), pretvorimo LP/PH v LP/PW (pari črt na širino slike) ob upoštevanju razmerja stranic. Standardni ISO grafikon ima razmerje stranic 4:3, zato je $\text{LP/PW} = \text{LP/PH} \times \frac{3}{4} = n \times 100 \times \frac{3}{4} = n \times 75$. Kvocient LP/PW uporabimo namesto LP/PH v enačbi (3.3), da dobimo pravilno pretvorbo za vertikalno ločljivost.

Slika 3.8 prikazuje potek modulacijske prenosne funkcije glede na izluščena podpodročja pri vaji 3.5. Iz slike ocenimo, da je polovična vrednost najvišjega kontrasta približno 0,9, kontrast pa

pade polovico maksimalne vrednosti pri oznaki $n=7$, kar pomeni: Primer za $n = 5$:

$$\text{LPPI}_7 = \frac{7 \times 100}{7,09} \approx 98,8 \text{ LPPI} \quad (3.4)$$

Za verifikacijo lahko neposredno izmerimo širino enega para črt (črna + bela črta) v milimetrih:

$$\text{LPPI} = \frac{25,4 \text{ mm/palec}}{\text{širina para črt v mm}} \quad (3.5)$$

Opomba: Če je grafikon pri tiskanju spremenil razmerje stranic, je potrebno uporabiti dejanske dimenzije in prilagoditi izračune. Pri merjenju ločljivosti optičnih sistemov je treba upoštevati tudi Nyquistov limit in možnost pojava Moiré vzorcev (tj. preslikovanja višjih frekvenc v nižje, kar povzroča artefakte v slikah).

Vaja 3.7 Napišite funkcijo za izračun efektivnega dinamičnega območja b_{ef} na sivinskih slikah kalibra ColorChecker iImages:

```
1 def computeEffDynRange( iImages ):
2     # Tu napišite python kodo funkcije
3     return oEDR
```

ki vrne skalarno vrednost oEDR. Vrednosti L_{min} in L_{max} določite kot povprečne vrednosti μ najtemnejše in najsvetlejše podslike, moč šuma σ_n pa kot povprečje standardnih deviacij v vseh podslikah. Preverite, ali je oEDR večje pri bolj osvetljenem kalibru. ■

Najprej za vsako podsliko izračunamo standardni odklon (σ_n), nato določi razliko med maksimalno in minimalno povprečno intenziteto podslik ter jo delimo s povprečjem standardnih odklonov (σ_{avg}) vseh podslik. Končni rezultat je logaritemska vrednost (osnova 2) razmerja teh dveh vrednosti, ki predstavlja število razločljivih stopenj intenzitete, ki jih sistem lahko reproducira. Matematično je EDR izražen kot $\text{EDR} = \log_2 \left(\frac{L_{max} - L_{min}}{\sigma_{avg}} \right)$. Višja vrednost EDR označuje boljše sposobnost sistema za zajemanje širokega razpona svetlosti in razlikovanja med nivoji svetlosti.

```
1 def computeEffDynRange(iImages):
2     '''
3     Izracunaj efektivno dinamicno obmocje
4
5     Parameters
6     -----
7     iImages : numpy.ndarray
8         Vhodne slike
9
10    Returns
11    -----
12    oEDR : float
13        Vrednost efektivnega dinamicnega obmocja
14    '''
15    L = np.zeros((len(iImages),))
16    sig = np.zeros((len(iImages),))
17    for i in range(len(iImages)):
18        L[i] = np.mean(iImages[i].flatten())
19        sig[i] = np.std(iImages[i].flatten())
```

```

20     oEDR = np.log2((L.max() - L.min())/sig.mean())
21     return oEDR
22
23 # test funkcije
24 print('Efektivno dinamično območje v bitih: {:.2f}'.format(
25     computeEffDynRange(sub_gray_svetla)))

```

Pri izvedbi gornje funkcije dobimo odgovor:

```
Efektivno dinamično območje v bitih: 6.61
```

kar pomeni, da pri trenutnih svetlobnih razmerah in nastavitvah kamere (zaslonka, čas ekspozicije) ni izkoriščeno celotno dinamično območje svetlobnega tipala, ki je 8 bitov.

Vaja 3.8 Napišite funkcijo za izračun diferencialnega razmerja signal–šum SNR_D na dveh poljubnih sivinskih slikah `iImage1` in `iImage2`:

```

1 def computeSNR( iImage1, iImage2 ):
2     # Tu napišite python kodo funkcije
3     return oSNR

```

ki vrne skalarno vrednost `oSNR`. Preizkusite delovanje funkcije tako, da izračunate SNR_D med pari podslik kalibra Colorchecker z indeksoma 19 in 24 in med 23 in 24. Komentirajte dobljene vrednosti. ■

Odločimo se za implementacijo diferencialnega razmerja signal–šum. Najprej izračunamo povprečni vrednosti (μ_1, μ_2) in standardna odklona (σ_1, σ_2) za obe vhodni (pod)sliki. Diferencialno razmerje signal/šum se nato določi kot absolutna razlika povprečij, deljena s kvadratnim korenom vsote kvadratov standardnih odklonov: $SNR_D = \frac{|\mu_1 - \mu_2|}{\sqrt{\sigma_1^2 + \sigma_2^2}}$. Višje vrednosti SNR_D označujejo boljšo ločljivost med signaloma v primerjavi s šumom oz. večjo razliko med obravnavanima slikama glede na njihovo variabilnost.

```

1 def computeSNR(iImage1, iImage2):
2     '''
3     Vrne diferencialno razmerje signal/sum
4
5     Parameters
6     -----
7     iImage1, iImage2 : np.ndarray
8         Sliki področij zanimanja, med katerima računamo SNR
9
10    Returns
11    -----
12    oSNR : float
13        Vrednost razmerja signal/sum
14    '''
15    mu1 = np.mean(iImage1.flatten())
16    mu2 = np.mean(iImage2.flatten())
17
18    sig1 = np.std(iImage1.flatten())
19    sig2 = np.std(iImage2.flatten())
20
21    oSNR = np.abs(mu1 - mu2)/np.sqrt(sig1**2 + sig2**2)
22

```

```

23     return oSNR
24
25 # test funkcije
26 print('SNR med področjema z indeksoma 19 in 24: {:.2f}'.format(
27     computeSNR(sub_gray_svetla[18], sub_gray_svetla[23])))
28 print('SNR med področjem z indeksoma 23 in 24: {:.2f}'.format(
29     computeSNR(sub_gray_svetla[23], sub_gray_svetla[23])))

```

Dobimo naslednji rezultat:

```

SNR med področjema z indeksoma 19 in 24: 83.30
SNR med področjem z indeksoma 23 in 24: 7.86

```

ki kaže da se najtemnejše področje pri indeksu 19 zelo dobro ločuje glede na najtemnejše področje ($SNR_D \gg 1$). Področji z indeksoma 23 in 24 se zaradi manjše razlike slabše, a še vedno očino ločita glede na intenziteto in stopnjo šuma.

3.4 Naloge in vprašanja

Poročilo naloge pripravite v obliki Python skripte, ki naj izvede zahtevane izračune oz. klice funkcij in izriše pripadajoče slike oz. grafe. Vaša dolžnost je, da v skripti uvozite ustrezne Python knjižnice in dodate morebitne podporne skripte. Skripta se mora izvesti in poustvariti rezultate, kot se zahtevajo v nalogah. Na vprašanja odgovorite v obliki komentarja, naprimer *# Naloga X.2: To je moj odgovor..*

Naloga 3.1 Izračunajte povprečne sivinske vrednosti μ_i področij $i = 19, \dots, 24$ v dveh zajetih slikah ColorChecker kalibra in jih kot dve krivulji vrišite v skupen graf, ki prikazuje odvisnost μ_i od pripadajočega indeksa področja i . Določite enačbo, ki opisuje oblike teh dveh krivulj in obrazložite kako se med seboj razlikujeta pri bolj oz. manj osvetljenem kalibru?

Naloga 3.2 Za izračun histograma sivinske slike uporabite funkcijo `histogram` v Python knjižnici `numpy`. Pokažite, da funkcija vrne enak rezultat kot funkcija `computeHistogram()`, ki smo jo napisali na laboratorijskih vajah.

Naloga 3.3 S pomočjo podslik kalibra ISO 12233 izračunajte modulacijski prenosni funkciji ločeno za vodoravne in navpične linije in ju kot dve krivulji vrišite v skupen graf. Iz grafov pri vrednosti $m_{50\%}$ določite prostorsko ločljivost vašega fotoaparata v vodoravni in navpični osi slike in jo izrazite v lp/mm. Upoštevajte, da ima kaliber višino $H = 145$ mm, pripadajoče oznake (1–10) pa označujejo mnogokratnik $100 \times$ število linij na višino slike H , iz česar lahko izračunate število parov linij na milimeter (lp/mm).

Naloga 3.4 Določite *maksimalno dosegljivo* vrednost efektivnega dinamičnega območja b_{ef} vašega fotoaparata na zajetih slikah. Navedite načine s katerimi bi lahko povečali b_{ef} .

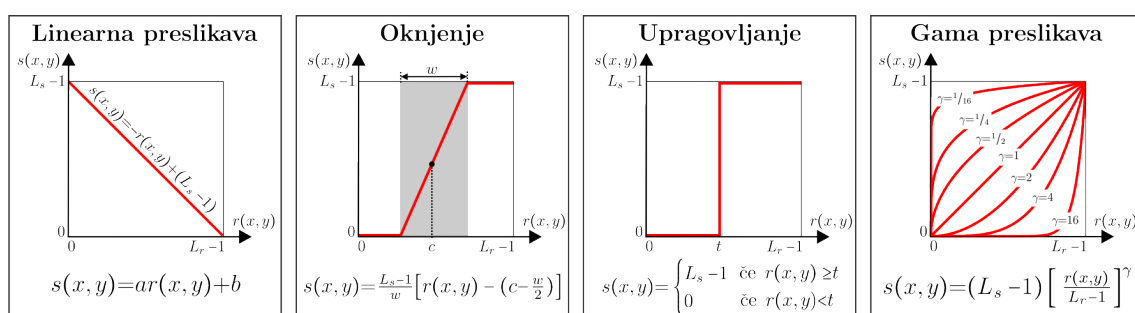
Naloga 3.5 Poiščite tisti par sivinskih podslik kalibra ColorChecker, pri katerem je SNR_D največji in navedite pripadajoče indekse področij (1–24). Ali bi lahko ti dve področji navedli brez izračuna SNR_D ? Odgovor obrazložite.

Naloga 3.6 Poiščite tiste pare sivinskih podslik kalibra ColorChecker, pri katerih je SNR_D manjši kot 1 in navedite pripadajoče indekse področij (1–24). Ali lahko na sivinski sliki kalibra ColorChecker med seboj razločite pare področij, ki imajo $SNR_D < 1$?

4. Preslikave sivin in barv

4.1 Sivinske preslikave

Sivinske preslikave so poljubne preslikave $\mathcal{T} : \mathbb{R} \rightarrow \mathbb{R}$ dinamičnega območja $[0, L_r - 1]$ referenčne slike $r(x, y)$ v dinamično območje $[0, L_s - 1]$ preslikane slike $s(x, y)$. Preslikavo izvedemo na vseh slikovnih elementih referenčne slike kot $s(x, y) = \mathcal{T}(r(x, y))$. Glavni namen sivinskih preslikav je **povečanje kontrasta** struktur zanimanja, uporabljajo pa se tudi za **prilagoditev dinamičnega območja** referenčne slike za potrebe prikazovanja. Pri vaji boste spoznali linearno preslikavo, oknjenje, upravljanje in nelinearno gama preslikavo (slika 4.1).



Slika 4.1: Sivinske preslikave kot homogene točkovne operacije, pri čemer je $r(x, y)$ vhodna, $s(x, y)$ pa izhodna slika.

4.2 Preslikave med barvnimi prostori

Barvo slikovnega elementa običajno definiramo s tremi, včasih pa celo z dvema komponentama oz. vrednostima. Zaradi načina pretvorbe svetlobe v digitalni zapis se najpogosteje uporablja zapis barve s komponentami *RGB*, ki ustrezajo trem različnim detektorjem svetlobe in ki so selektivno občutljivi pri valovnih dolžinah svetlobe okoli 700 nm (*R*), 550 nm (*G*) in 450 nm (*B*). Obstajajo tudi drugi barvni prostori, ki so bolj primerni za analizo digitalnih slik, naprimer *HSV* in $L^*a^*b^*$.

Sliko v RGB barvnem prostoru lahko pretvorimo v drug barvni prostor z (ne)linearno preslikavo RGB komponent.

Preslikava iz RGB v XYZ barvni prostor pri referenčni beli osvetlitvi D65 je definirana kot:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0,4125 & 0,3576 & 0,1804 \\ 0,2127 & 0,7151 & 0,0722 \\ 0,0193 & 0,1192 & 0,9505 \end{bmatrix} \begin{bmatrix} R \\ B \\ G \end{bmatrix}, \quad (4.1)$$

pri čemer morajo vrednosti komponent R , G in B ležati na intervalu $[0, 1]$. Preslikavi iz barvnega prostora XYZ v barvni prostor $L^*a^*b^*$ in obratno sta definirana kot:

$$\begin{aligned} L^* &= 116Y^{1/3} - 16 & L^* &= 116Y^{1/3} - 16 \\ a^* &= 500(X^{1/3} - Y^{1/3}) & a^* &= 500(X^{1/3} - Y^{1/3}) \\ b^* &= 200(Y^{1/3} - Z^{1/3}) & b^* &= 200(Y^{1/3} - Z^{1/3}) \end{aligned} \quad (4.2)$$

Barvni prostor $L^*a^*b^*$ je zaradi linearne metrike med različnimi barvnimi odtenki primeren za kvantitativno primerjavo in analizo barv. Spodaj je prikazana preslikava iz RGB v HSV barvni prostor ter pripadajoča inverzna preslikava.

Pretvorba iz RGB v HSV	Pretvorba iz HSV v RGB
Predizračun: $r = R/255, g = G/255, b = B/255$ $C_{max} = \max(r, g, b)$ $C_{min} = \min(r, g, b)$ $\Delta = C_{max} - C_{min}$ Barvni odtenek oz. <i>hue</i> $H \in [0^\circ, 360^\circ]$: $H = \begin{cases} 0^\circ \times \left(\frac{g-b}{\Delta} \bmod 6\right) & \text{če } \Delta \equiv 0 \\ 60^\circ \times \left(\frac{g-b}{\Delta} \bmod 6\right) & \text{če } C_{max} \equiv r \\ 60^\circ \times \left(\frac{b-r}{\Delta} + 2\right) & \text{če } C_{max} \equiv g \\ 60^\circ \times \left(\frac{r-g}{\Delta} + 4\right) & \text{če } C_{max} \equiv b \end{cases}$ Barvno nasičenje oz. <i>saturation</i> $S \in [0, 1]$: $S = \begin{cases} 0 & \text{če } \Delta = 0 \\ \frac{\Delta}{C_{max}} & \text{sicer} \end{cases}$ Svetlost oz. <i>value</i> $V \in [0, 1]$: $V = C_{max}$	Predizračun: $C = V \times S$ $X = C \times (1 - H/60^\circ \bmod 2 - 1)$ $m = V - C$ Komponente (r, g, b) določimo glede na vrednost $H \in [0^\circ, 360^\circ]$: $(r, g, b) = \begin{cases} (C, X, 0) & \text{če } 0^\circ \leq H < 60^\circ \\ (X, C, 0) & \text{če } 60^\circ \leq H < 120^\circ \\ (0, C, X) & \text{če } 120^\circ \leq H < 180^\circ \\ (0, X, C) & \text{če } 180^\circ \leq H < 240^\circ \\ (X, 0, C) & \text{če } 240^\circ \leq H < 300^\circ \\ (C, 0, X) & \text{če } 300^\circ \leq H < 360^\circ \end{cases}$ Komponente RGB nato dobimo z: $(R, G, B) = (r + m, g + m, b + m)$ Izhodne RGB vrednosti so na intervalu $[0, 1]$.

4.3 Vaje z rešitvami

Pri vaji boste napisali funkcije sivinskih preslikav in funkcije za pretvorbo RGB slik v HSV barvni prostor ter preizkusili teh funkcij delovanje na sivinski in barvni sliki. Koda vaje je dostopna na

[Git repozitoriju](#).

Vaja 4.1 Naložite barvno RGB sliko `slika.jpg` v okolje Python z ukazom `open()` v knjižnici `PIL.Image` in jo pretvorite v 8-bitno sivinsko sliko kot $S = 0,299R + 0,587G + 0,114B$. Poskrbite za ustrezno zaokroževanje in pretvorbo tipa sivinskih vrednosti slikovnih elementov. ■

Najprej uvozimo knjižnice, ki jih bomo potrebovali pri vaji:

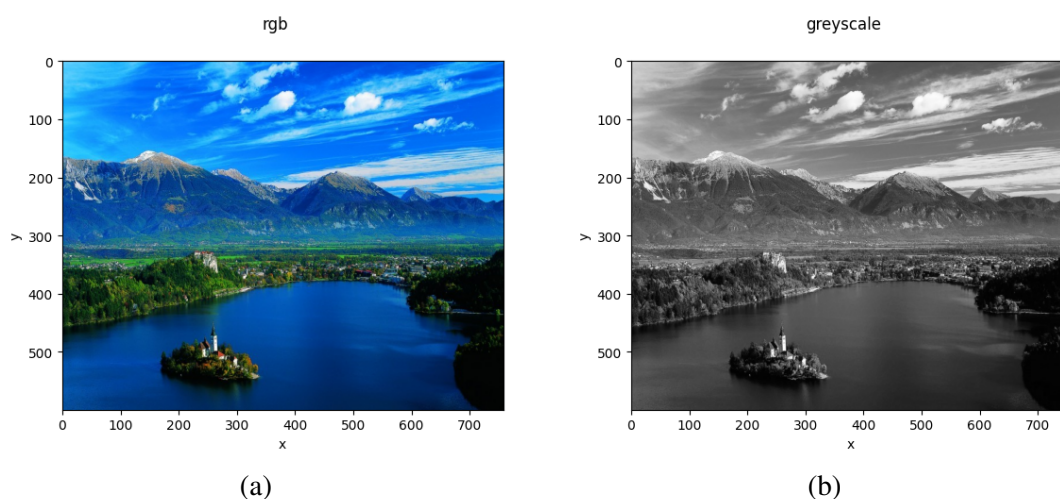
```
1 import cvlib
2 import matplotlib.pyplot as plt
3 import numpy as np
4 import PIL.Image as im
```

Opomba: uvozili smo knjižnico `cvlib`, v kateri so funkcije napisane na predhodnih vajah. Na primer, `loadImage()`, `showImage()` in `saveImage()`.

Nalaganje in prikaz slik izvedemo s kodo:

```
1 def colorToGray(iImage):
2     dtype = iImage.dtype # zapomnimo si tip zapisa vhodne slike
3     rgb = np.array(iImage, dtype='float') # pretvorimo v zapis s plavajočo vejico
4     return (rgb[:, :, 0]*0.299 + rgb[:, :, 1]*0.587 +
5             rgb[:, :, 2]*0.114).astype(dtype) # preslikavo in pretvorimo v originalni
6                                             zapis
7
8 iImage = loadImage('slika.jpg') # nalaganje slike
9 iImageG = colorToGray( iImage ) # pretvorba slike
10 cvlib.showImage( iImage, 'rgb' ) # prikaz originalne in
11 cvlib.showImage( iImageG, 'greyscale' ) # preslikane slike
```

Slika 4.2 prikazuje originalno RGB in pripadajočo sivinsko sliko.



Slika 4.2: Slika (a) prikazuje dano RGB sliko `slika.jpg`, slika (b) pa pripadajočo sivinsko sliko.

Vaja 4.2 Napišite funkcijo za poljubno linearno sivinsko preslikavo sivinske slike `iImage`:

```
1 def scaleImage( iImage, iSlopeA, iIntersectionB ):
2     # Tu napišite python kodo funkcije
```

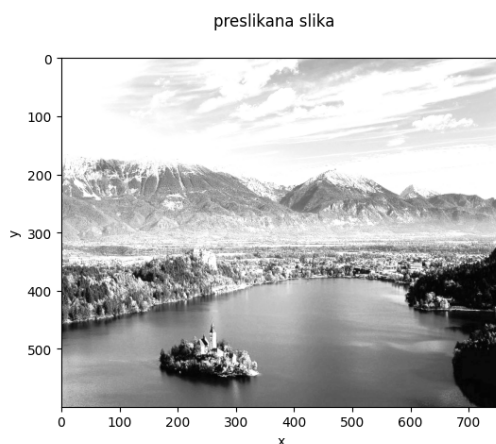
```
3 | return oImage
```

kjer sta $iSlopeA$ in $iIntersectionB$ parametra a in b linearne preslikave. Funkcija vrne linearno preslikano sivinsko sliko $oImage$, ki naj bo enakega tipa kot vhodna slika. Minimalne in maksimalne vrednosti celoštevilskih podatkovnih tipov dobite z `numpy.iinfo( dtype ).min` in `numpy.iinfo( dtype ).max`. Preverite delovanje funkcije na sivinski sliki s poljubno linearno sivinsko preslikavo. ■

Funkcija `scaleImage(iImage, iSlopeA, iIntersectionB)` izvede linearno preslikavo sivinskih vrednosti slike po formuli $s(x,y) = a \cdot r(x,y) + b$, kjer a predstavlja naklon ($iSlopeA$) in b presečišče ($iIntersectionB$), pri čemer sta a, b poljubni skalarni vrednosti:

```
1 | def scaleImage( iImage, iSlopeA, iIntersectionB ):
2 |     """Linearna sivinska preslikava: a*I+b"""
3 |     iImageType = iImage.dtype
4 |     iImage = np.array( iImage, dtype='float' )
5 |     oImage = iSlopeA * iImage + iIntersectionB # linearna preslikava
6 |     # zaokrozevanje vrednosti
7 |     if iImageType.kind in ('u','i'):
8 |         oImage[oImage<np.iinfo(iImageType).min] = np.iinfo(iImageType).min
9 |         oImage[oImage>np.iinfo(iImageType).max] = np.iinfo(iImageType).max
10 |    return np.array( oImage, dtype=iImageType )
11 |
12 | # preizkus funkcije
13 | iImageG = colorToGray( iImage )
14 | oImageG = scaleImage( iImageG, 2.0, 32 )
15 | rvlb.showImage( oImageG, 'preslikana slika' )
```

Poleg preslikave poskrbimo za pravilno zaokroževanje preslikanih sivinskih vrednosti. Najprej pretvorimo vhodno sliko v tip `float`, nato uporabi linearno transformacijo in na koncu zagotovi pravilno zaokrožitev ter omejitev vrednosti znotraj obsega originalnega podatkovnega tipa. Če je vhodna slika celoštevilska (`uint8`), funkcija skrbi, da so vse vrednosti v mejah $[0, 255]$. Rezultat je normalizirana sivinska slika v originalnem formatu, primerna za nadaljnjo obdelavo ali prikaz. Slika 4.3 prikazuje preslikano sivinsko sliko 4.2b.



Slika 4.3: Linearno preslikana sivinska slika 4.2b, kjer smo uporabili $a = 2$ (povečanje kontrasta) in $b = 32$ (posvetlitev).

Vaja 4.3 Napišite funkcijo za poljubno linearno sivinsko oknjenje sivinske slike iImage:

```
1 def windowImage( iImage, iCenter, iWidth ):
2     # Tu napišite python kodo funkcije
3     return oImage
```

kjer sta iCenter in iWidth pa parametra c in w linearnega oknjenja. Funkcija vrne linearno sivinsko oknjeno oImage, ki naj bo enakega tipa kot vhodna slika. Minimalne in maksimalne vrednosti celoštevilskih podatkovnih tipov dobite z `numpy.iinfo( dtype ).min` in `numpy.iinfo( dtype ).max`. Preverite delovanje funkcije na sivinski sliki s poljubnim sivinskim oknjenjem. ■

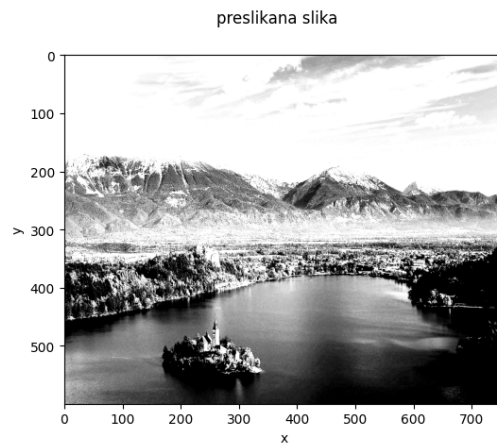
Funkcija `windowImage(iImage, iCenter, iWidth)` izvaja linearno oknjenje slikovnih podatkov, kar je postopek, ki preslika območje intenzitet v okolici središča $iCenter$ s širino $iWidth$ na celotno razpoložljivo območje vrednosti. Algoritem najprej določi dinamično območje vhodne slike (razlikuje med celoštevilskimi in realnimi podatkovnimi tipi), nato izračuna naklon $iSlopeA$ in presečišče $iInterceptB$ za linearno transformacijo po formuli $s(x,y) = \frac{(L_s-1)}{w} \cdot (r(x,y) - (c - \frac{w}{2}))$, kjer L_s predstavlja območje izhodnih vrednosti, w širino okna in c središče okna. Gre torej le za linearno preslikavo, ki ima drugačen pomen parametrov. Rešitev lahko zapišemo:

```
1 def windowImage( iImage, iCenter, iWidth ):
2     """Linearno oknjenje: (Ls-1)/w*(I-(c-w/2))"""
3     iImageType = iImage.dtype
4     if iImageType.kind in ('u','i'):
5         iMaxValue = np.iinfo(iImageType).max
6         iMinValue = np.iinfo(iImageType).min
7         iRange = iMaxValue - iMinValue
8     else:
9         iMaxValue = np.max( iImage )
10        iMinValue = np.min( iImage )
11        iRange = iMaxValue - iMinValue
12        # preračunamo koeficiente linearne preslikave
13        iSlopeA = iRange / float(iWidth)
14        iInterceptB = - iSlopeA * ( float(iCenter) - iWidth / 2.0 )
15        # izvedemo linearno preslikavo
16        return scaleImage( iImage, iSlopeA, iInterceptB )
17
18 # preizkus funkcije
19 iImageG = colorToGray( iImage )
20 oImageG = windowImage( iImageG, 64, 64 )
21 rvlb.showImage( oImageG, 'preslikana slika')
```

Končni rezultat je preslikana sivinska slika, kjer so vrednosti zunaj izbranega območja oknjenja obrezane na minimalno ali maksimalno dovoljeno vrednost. Slika 4.4 prikazuje preslikano sivinsko sliko 4.2b.

Vaja 4.4 Napišite funkcijo za poljubno sivinsko upravljanje sivinske slike iImage:

```
1 def thresholdImage( iImage, iThreshold ):
2     # Tu napišite python kodo funkcije
3     return oImage
```

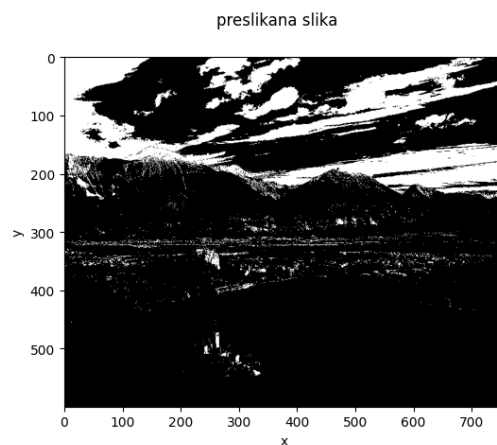
Slika 4.4: Oknjena sivinska slika 4.2b, kjer smo uporabili $c = 64$ (center okna) in $w = 64$ (širina okna).

kjer je `iThreshold` parameter upravljanja oz. prag t . Funkcija vrne upravljenjo sliko `oImage`. Preverite delovanje funkcije na sivinski sliki s poljubnim pragom t . ■

Vhodno sliko pretvorimo v binarno sliko glede na izbrano prazno vrednost `iThreshold` oz. t , pri čemer vrednosti slike $r(x,y) > t$ postavimo na vrednost 255, ostale pa na vrednost 0. Rešitev lahko zapišemo:

```
1 def thresholdImage( iImage, iThreshold ):
2     """Upravljanje: I>t"""
3     # iImage = np.asarray( iImage )
4     oImage = 255 * np.array(iImage>iThreshold, dtype='uint8')
5     return oImage
6
7 # preizkus funkcije
8 iImageG = colorToGray( iImage )
9 oImageG = thresholdImage( iImageG, 128 )
10 rvlb.showImage( oImageG, 'preslikana slika')
```

Končni rezultat je binarna slika kot prikazuje slika 4.5.



Slika 4.5: Upravljen svin slika 4.2b, kjer smo uporabili $t = 128$.

Vaja 4.5 Napišite funkcijo za poljubno gama sivinsko preslikavo sivinske slike iImage:

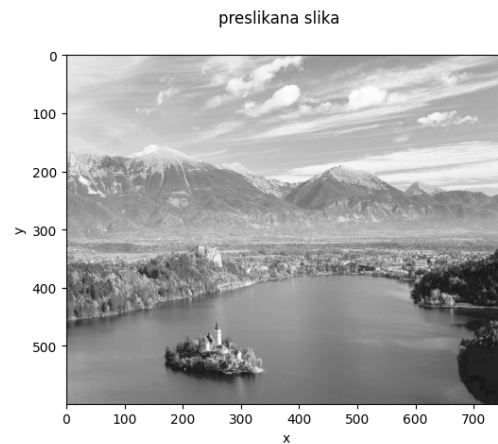
```
1 def gammaImage( iImage, iGamma ):
2     # Tu napišite python kodo funkcije
3     return oImage
```

kjer je iGamma parameter γ . Funkcija vrne gama sivinsko preslikano sliko oImage, ki naj bo enakega tipa kot vhodna slika. Preverite delovanje funkcije na sivinski sliki z $\gamma > 1$ in $\gamma < 1$ in analizirajte vpliv na histogram slike. ■

Funkcija gammaImage(iImage, iGamma) izvaja nelinearno gama korekcijo slike po formuli $s(x,y) = (L_s - 1) \left(\frac{r(x,y)}{L_r - 1} \right)^\gamma$, kjer $r(x,y)$ predstavlja vhodne vrednosti piksla, $s(x,y)$ izhodne vrednosti, L_s območje izhodnih vrednosti, L_r območje vhodnih vrednosti in γ eksponent korekcije. Algoritem najprej normalizira vrednosti slike v območje [0,1], nato uporabi potenčno preslikavo z eksponentom iGamma, nakar denormalizira rezultat nazaj v originalno območje vrednosti. Za celoštevilске podatkovne tipe zagotovi pravilno zaokroževanje in omejitev vrednosti znotraj dovoljenega območja. Rešitev lahko zapišemo:

```
1 def gammaImage( iImage, iGamma ):
2     """ Gama preslikava: (Ls-1)(I/(Lr-1))^gamma """
3     iImageType = iImage.dtype
4     iImage = np.array( iImage, dtype='float' )
5     # preberi mejne vrednosti in območje vrednosti
6     if iImageType.kind in ('u','i'):
7         iMaxValue = np.iinfo(iImageType).max
8         iMinValue = np.iinfo(iImageType).min
9         iRange = iMaxValue - iMinValue
10    else:
11        iMaxValue = np.max( iImage )
12        iMinValue = np.min( iImage )
13        iRange = iMaxValue - iMinValue
14    # izvedi gamma preslikavo
15    iImage = (iImage - iMinValue) / float(iRange)
16    oImage = iImage ** iGamma
17    oImage = float(iRange) * oImage + iMinValue
18    # zaokrozevanje vrednosti
19    if iImageType.kind in ('u','i'):
20        oImage[oImage<np.iinfo(iImageType).min] = np.iinfo(iImageType).min
21        oImage[oImage>np.iinfo(iImageType).max] = np.iinfo(iImageType).max
22    # vrni sliko v originalnem formatu
23    return np.array( oImage, dtype=iImageType )
24
25 # preizkus funkcije
26 iImageG = colorToGray( iImage )
27 oImageG = gammaImage( iImageG, 1/2 )
28 rvlb.showImage( oImageG, 'preslikana slika')
```

Gama korekcija omogoča prilagoditev kontrasta v različnih delih svetlosti, pri čemer vrednosti $\gamma < 1$ povečajo kontrast v temnih predelih, vrednosti $\gamma > 1$ pa v svetlih predelih slike. Za primer $\gamma = \frac{1}{2}$ je končni rezultat sivinska slika s povečanim kontrastom na temnih delih originalne sivinske slike (slika 4.5); povečanje kontrasta je očitno iz pripadajočega histograma na sliki 4.7b, ki je v primerjavi s tistim na sliki 4.7a, razredčen na področju nizkih intenzitet – to pomeni, da poveča kontrast med nizkimi sivinskimi vrednostmi.

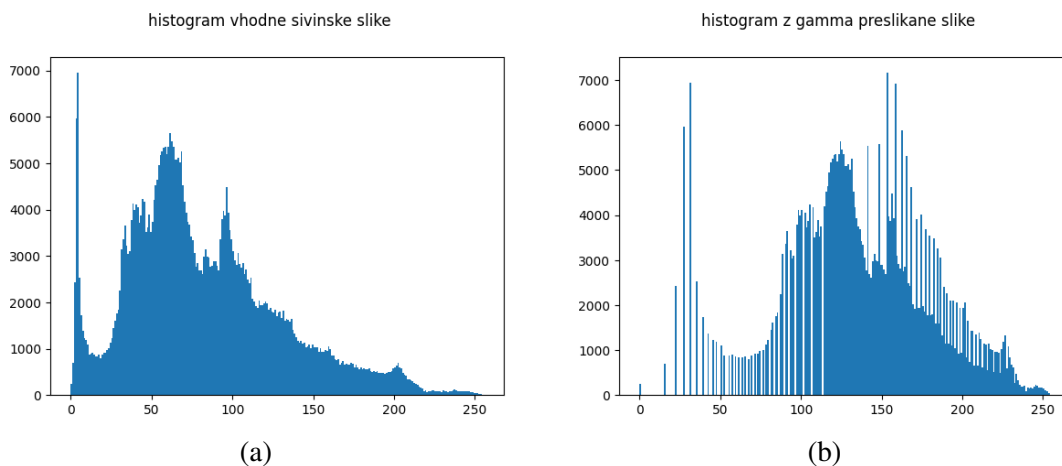


Slika 4.6: Preslikana sivinska slika 4.2b z $\gamma = \frac{1}{2}$.

Za dani primer si oglejmo še histograma slik pred in po preslikavi, z uporabo kode:

```
1 plt.figure()
2 plt.hist(iImageG.flatten(),255, range=(0,255))
3 plt.suptitle('histogram vhodne sivinske slike')
4 plt.figure()
5 plt.hist(oImageG.flatten(),255, range=(0,255))
6 plt.suptitle('histogram z gamma preslikane slike')
```

Histograma sta prikazana na sliki 4.7.



Slika 4.7: Slika (a) prikazuje histogram vhodne sivinske slike `slika.jpg` na sliki 4.2b, slika (b) pa histogram z `gammaImage()` preslikano sivinsko sliko.

Vaja 4.6 Napišite funkcijo za preslikavo barvne slike `iImage` med *RGB* in *HSV* barvnimi prostori:

```
1 def convertImageColorSpace( iImage, iConversionType ):
2     # Tu napišite python kodo funkcije
```

```
3 | return oImage
```

kjer `iConversionType` določa pretvorbo iz izvirnega v ciljni barvni prostor in lahko zavzame vrednosti `'RGBtoHSV'` ali `'HSVtoRGB'`. Preverite delovanje funkcije s pretvorbo barvne slike iz `RGB` v `HSV` prostor in nato nazaj iz `HSV` v `RGB`. Dobljena slika mora biti enaka vhodni sliki. ■

Funkcija `convertImageColorSpace(iImage, iConversionType)` izvaja dvosmerno pretvorbo med barvnima prostoroma `RGB` in `HSV`. Pri pretvorbi iz `RGB` v `HSV` (`'RGBtoHSV'`) algoritem najprej normalizira vrednosti v območje `[0,1]`, nato izračuna komponente `HSV` po standardnih formulah: barvni odtenek (`H`) se določi glede na dominantno barvo, nasičenost (`S`) kot razmerje med razliko največje in najmanjše komponente ter največjo komponento, svetlost (`V`) pa ustreza največji vrednosti med `RGB` komponentami. Obratna pretvorba (`'HSVtoRGB'`) uporablja segmentirano logiko za pretvorbo `H` v ustrezen sektor barvnega kolesa, nato pa izračuna ustrezne `RGB` vrednosti z upoštevanjem nasičenosti in svetlosti. Funkcija avtomatsko upravlja z mejnimi vrednostmi in zagotavlja pravilno območje vrednosti za vsak barvni prostor, pri čemer ohranja natančnost barvnega podajanja. Rešitev lahko zapišemo:

```
1 | def convertImageColorSpace( iImage, iConversionType ):
2 |     """Pretvorba barvne slike med barvnima prostoroma RGB in HSV"""
3 |     iImage = np.array( iImage, dtype='float' )
4 |
5 |     if iConversionType == 'RGBtoHSV':
6 |         iImage = iImage / 255.0
7 |         r, g, b = iImage[:, :, 0], iImage[:, :, 1], iImage[:, :, 2]
8 |
9 |         h = np.zeros_like( r )
10 |        s = np.zeros_like( r )
11 |        v = np.zeros_like( r )
12 |
13 |        Cmax = np.maximum(r, np.maximum(g, b))
14 |        Cmin = np.minimum(r, np.minimum(g, b))
15 |        delta = Cmax - Cmin + 1e-7
16 |
17 |        h[Cmax == r] = 60.0 * ((g[Cmax == r] - b[Cmax == r])/delta[Cmax == r] %
18 |                               6.0)
19 |        h[Cmax == g] = 60.0 * ((b[Cmax == g] - r[Cmax == g])/delta[Cmax == g] +
20 |                               2.0)
21 |        h[Cmax == b] = 60.0 * ((r[Cmax == b] - g[Cmax == b])/delta[Cmax == b] +
22 |                               4.0)
23 |
24 |        s[delta!=0.0] = delta[delta!=0.0] / (Cmax[delta!=0.0] + 1e-7)
25 |
26 |        v = Cmax
27 |
28 |        # ustvari izhodno sliko
29 |        oImage = np.zeros_like( iImage )
30 |        oImage[:, :, 0] = h; oImage[:, :, 1] = s; oImage[:, :, 2] = v;
31 |
32 |        return oImage
33 |
34 |     elif iConversionType == 'HSVtoRGB':
35 |         h = iImage[:, :, 0]; s = iImage[:, :, 1]; v = iImage[:, :, 2];
36 |
37 |         C = v * s
```

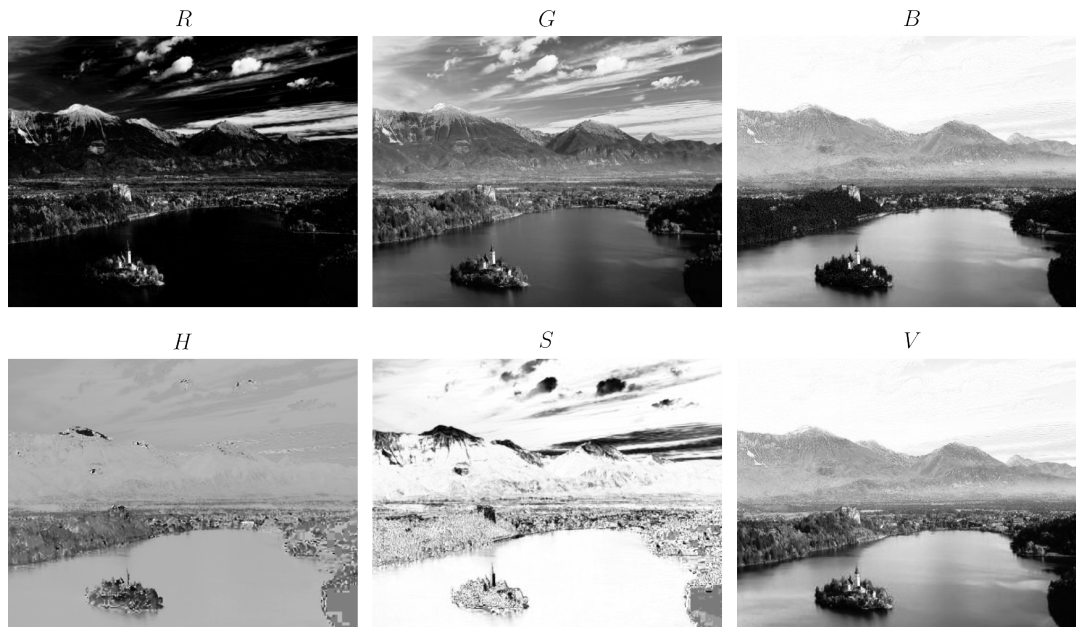
```

35     X = C * (1.0 - np.abs( ( h/60.0 ) % 2.0 ) - 1 ) )
36     m = v - C
37
38     r = np.zeros_like( h )
39     g = np.zeros_like( h )
40     b = np.zeros_like( h )
41
42     r[ (h>=0.0) * (h<60.0) ] = C[ (h>=0.0) * (h<60.0) ]
43     g[ (h>=0.0) * (h<60.0) ] = X[ (h>=0.0) * (h<60.0) ]
44
45     r[ (h>=60.0) * (h<120.0) ] = X[ (h>=60.0) * (h<120.0) ]
46     g[ (h>=60.0) * (h<120.0) ] = C[ (h>=60.0) * (h<120.0) ]
47
48     g[ (h>=120.0) * (h<180.0) ] = C[ (h>=120.0) * (h<180.0) ]
49     b[ (h>=120.0) * (h<180.0) ] = X[ (h>=120.0) * (h<180.0) ]
50
51     g[ (h>=180.0) * (h<240.0) ] = X[ (h>=180.0) * (h<240.0) ]
52     b[ (h>=180.0) * (h<240.0) ] = C[ (h>=180.0) * (h<240.0) ]
53
54     r[ (h>=240.0) * (h<300.0) ] = X[ (h>=240.0) * (h<300.0) ]
55     b[ (h>=240.0) * (h<300.0) ] = C[ (h>=240.0) * (h<300.0) ]
56
57     r[ (h>=300.0) * (h<360.0) ] = C[ (h>=300.0) * (h<360.0) ]
58     b[ (h>=300.0) * (h<360.0) ] = X[ (h>=300.0) * (h<360.0) ]
59
60     r = r + m
61     g = g + m
62     b = b + m
63
64     # ustvari izhodno sliko
65     oImage = np.zeros_like( iImage )
66     oImage[:, :, 0] = r; oImage[:, :, 1] = g; oImage[:, :, 2] = b;
67
68     # zaokrozevanje vrednosti
69     oImage = 255.0 * oImage
70     oImage[oImage>255.0] = 255.0
71     oImage[oImage<0.0] = 0.0
72
73     oImage = np.array( oImage, dtype='uint8' )
74
75     return oImage
76
77 # preizkus funkcije
78 iImage = loadImage('slika.jpg')
79 oImageHSV = convertImageColorSpace( iImage, 'RGBtoHSV' )
80 oImageRGB = convertImageColorSpace( oImageHSV, 'HSVtoRGB' )
81
82 # prikaz HSV slike po komponentah
83 rvlb.showImage( oImageHSV[:, :, 0]*255.0/360.0, 'H slika' )
84 rvlb.showImage( oImageHSV[:, :, 1]*255.0, 'S slika' )
85 rvlb.showImage( oImageHSV[:, :, 2]*255.0, 'V slika' )
86
87 # prikaz RGB slike po komponentah
88 rvlb.showImage( oImageRGB[:, :, 0], 'R slika' )
89 rvlb.showImage( oImageRGB[:, :, 1], 'G slika' )

```

```
90 | rvlb.showImage( oImageRGB[:, :, 2], 'B slika')
```

Rezultat pretvorbe prikazuje slika 4.8, pri čemer so posamezni barvni kanali prikazani kot sivinske slike z zalogo vrednosti [0, 255].



Slika 4.8: Prikaz barvne slike 4.2a po komponentah RGB in HSV.

4.4 Naloge in vprašanja

Poročilo naloge pripravite v obliki Python skripte, ki naj izvede zahtevane izračune oz. klice funkcij in izriše pripadajoče slike oz. grafe. Vaša dolžnost je, da v skripti uvozite ustrezne Python knjižnice in dodate morebitne podporne skripte. Skripta se mora izvesti in poustvariti rezultate, kot se zahtevajo v nalogah. Na vprašanja odgovorite v obliki komentarja, naprimer

```
# Naloga X.2: To je moj odgovor..
```

Naloga 4.1 Določite vrednosti $iSlope$ (a) ter $iIntersection$ (b) linearne sivinske preslikave tako, da se bo najsvetlejša točka sivinske slike preslikala v najtemnejšo točko slike in nasprotno. Sliko prikažite.

Naloga 4.2 Določite vrednosti $iCenter$ (c) in $iWidth$ (w) sivinskega oknjenja tako, da boste iz sivinske slike odstranili 10% najtemnejših in 10% najsvetlejših slikovnih elementov. Oknjeno sliko prikažite.

Naloga 4.3 Določite vrednost $iThreshold$ (t) funkciji upravljanja tako, da bo natanko 50% najsvetlejših slikovnih elementov v vhodni sivinski sliki nad tem pragom. Upragovljeno sliko prikažite.

Naloga 4.4 Obrazložite vpliv vrednosti $iGamma$ (γ) v gama sivinski preslikavi na kontrast izhodne sivinske slike v primeru, ko je $\gamma < 1$ in v primeru, ko je $\gamma > 1$. Prikažite ustrezni sliki za oba primera.

Naloga 4.5 Napišite funkcijo za gama sivinsko preslikavo RGB barvne slike tako, da sliko najprej pretvorite v HSV barvni prostor, nato izvedite gama preslikavo nad komponento V , in HSV sliko

pretvorite nazaj v *RGB* barvni prostor. Prikažite primera gama preslikanih barvnih slik za $\gamma < 1$ in $\gamma > 1$. Ali ima tak način gama preslikave barvne slike enak vpliv na kontrast kot v primeru sivinske slike?

Naloga 4.6 Napišite funkcijo za preslikavo barvne slike v sivinsko sliko tako, da v *HSV* barvnem prostoru komponento barvnega nasičenja postavite na 0. Prikažite pridobljeno sivinsko sliko.

Naloga 4.7 Potrdite, da v barvni sliki `slika.jpg` prevladuje modra barva tako, da izračunate in izpišete razmerje med številom pikslov, ki predstavljajo modre odtenke, in številom vseh pikslov na sliki. Za modre odtenke vzemite piksle katerih vrednost *H* komponente je na območju 240 ± 30 stopinj.



5. Osnovna obdelava slik

Vaja je namenjena spoznavanju in razumevanju osnovnih postopkov za obdelavo slik kot so filtriranje, glajenje in ostrenje ter interpolacija in decimacija slik.

5.1 Filtriranje slik z diskretno konvolucijo

Filtriranje slike lahko izvedemo s postopkom **2D diskretne konvolucije** med podano sliko $g(x, y)$ in konvolucijskim jedrom $k(u, v)$ velikosti $U \times V$:

$$f(x, y) = \sum_{u=-U/2}^{U/2} \sum_{v=-V/2}^{V/2} k(u, v) g(x - u, y - v), \quad (5.1)$$

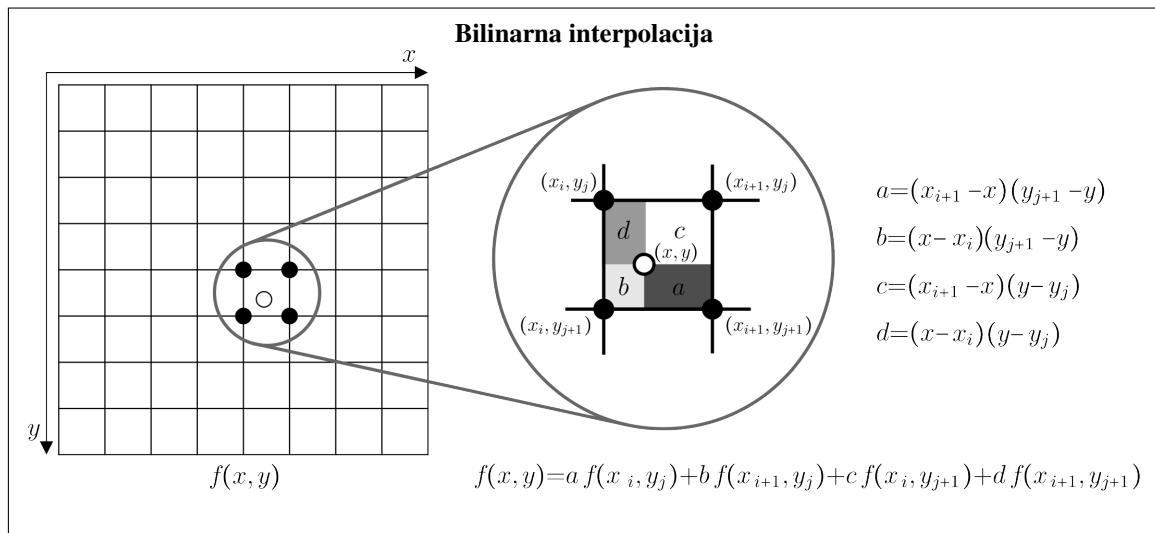
kjer je $f(x, y)$ izhodna slika. Na področju, kjer slika $g(x, y)$ ni definirana v skladu z definicijo konvolucije predpostavimo sivinsko vrednost 0. V Pythonu je 2D diskretno konvolucijo mogoče izračunati s štirimi vgnezenimi **for** zankami. Zunanji dve zanki uporabimo za naslavljanje slikovnih elementov slike $g(x, y)$, notranji dve zanki pa za naslavljanje konvolucijskega jedra $k(u, v)$. Konvolucijsko jedro $k(u, v)$ je definirano kot 2D matrika, središče jedra $k(0, 0) \leftrightarrow K$ pa v Pythonu ustreza indeksoma `floor(K.shape/2)`, kar je potrebno upoštevati pri naslavljanju 2D matrike konvolucijskega jedra.

5.2 Interpolacija slik

S postopkom **interpolacije slik** lahko priredimo sivinsko vrednost poljubni točki v slikovni ravnini. Na ta način lahko povečamo vzorčno frekvenco in s tem velikost slik ter tako navidežno zmanjšamo velikost slikovnih elementov. Glede na to, koliko sosednjih slikovnih elementov upoštevamo pri izračunu sivinske vrednosti v podani točki, delimo način interpolacije slik na:

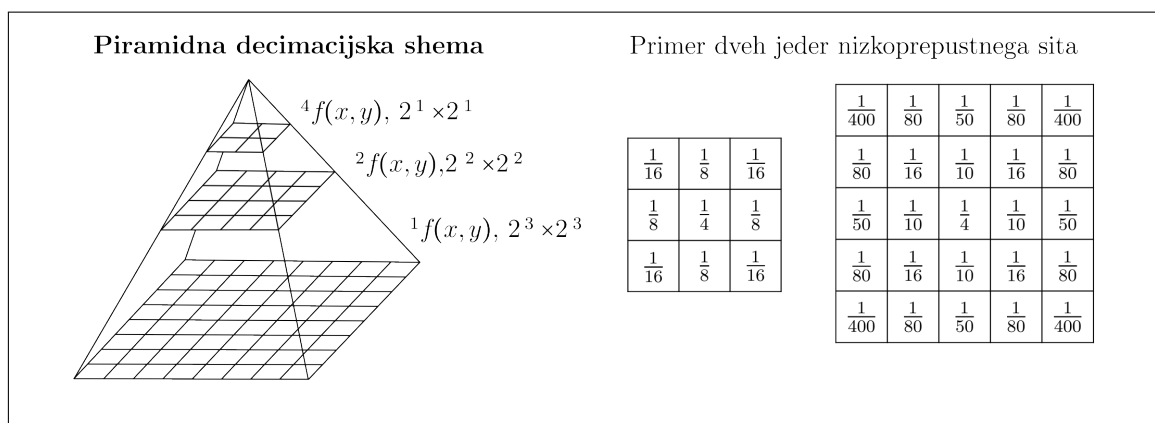
1. **ničti red** ali interpolacija najbližjega sosedu – upoštevamo le najbližji slikovni element,
2. **prvi red ali (bi)linearna interpolacija** – upoštevamo le štiri sosednje slikovne elemente,
3. **višji red**, npr. (bi)kubična interpolacija (drugi red), ki upošteva 16 sosednjih slikovnih elementov.

Računska zahtevnost interpolacijskih postopkov 2D slik v grobem raste s kvadratom reda interpolacije, kar pomeni, da je bikubična interpolacija (drugi red) približno štiri krat bolj zahtevna oziroma počasnejša od bilinearne interpolacije (prvi red). Naštete interpolacijske postopke je mogoče posplošiti, tako da delujejo tudi za večrazsežne slike.



5.3 Decimacija slik

S postopkom **decimacije slik** zmanjšamo vzorčno frekvenco ter s tem velikost slik. Skladno z Nyquistovim vzorčnim teoremom je pred postopkom decimacije potrebno sliko filtrirati z nizko prepustnim sitom in na ta način odstraniti visoko frekvenčno informacijo. Pri decimaciji se pogosto uporablja piramidna shema, kjer se vzorčna frekvenca izvirne slike zaporedoma zmanjšuje s celoštevilsko vrednostjo, običajno dva.



5.4 Vaje z rešitvami

Pri vaji boste napisali funkcije za filtriranje slik z 2D konvolucijo in preizkusili delovanje z različnimi jedri za namen glajenja in ostrenja sivinskih in barvnih slik. Koda vaje je dostopna na [Git repozitoriju](#). Funkcije za interpolacijo in decimacijo slik boste uporabili za povečavo oziroma pomanjšavo sivinskih in barvnih slik. Naložite barvno *RGB* sliko *slika.jpg* in jo

pretvorite v sivinsko sliko po enačbi $S = 0,299R + 0,587G + 0,114B$. Sliko uporabite za testiranje pravilnosti delovanja funkcij.

Vaja 5.1 Napišite funkcijo, ki izračuna 2D diskretno konvolucijo med vhodno sivinsko sliko `iImage` in konvolucijskim jedrom `iKernel`:

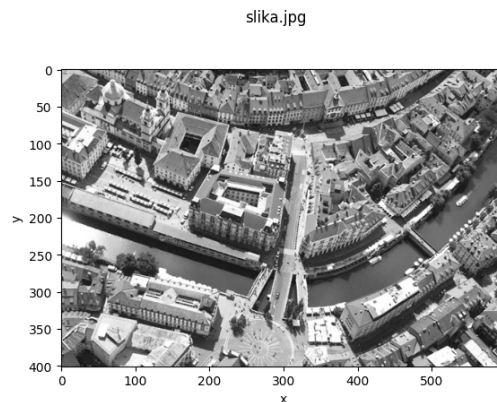
```
1 def discreteConvolution2D( iImage, iKernel ):
2     # Tu napišite python kodo funkcije
3     return oImage
```

kjer je `oImage` izhodna sivinska slika. Preizkusite delovanje funkcije 2D diskretne konvolucije in preverite njen vpliv na vhodno sivinsko sliko, tako da uporabite konvolucijsko jedro `iKernel = numpy.ones([k,k])/k**2` za različne vrednosti $k = 2n + 1, n = 1, 2, 3, \dots$ ■

Naložimo knjižnice, nato sliko `slika.jpg` in jo pretvorimo v sivinsko:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import cvlib
4
5 iImage = cvlib.loadImage( 'slika.jpg' )
6 iImageG = cvlib.colorToGray( iImage )
7 cvlib.showImage( iImageG, 'slika.jpg' )
```

Rezultat je prikazan na sliki 5.1.



Slika 5.1: Sivinska slika za analizo.

Funkcija `discreteConvolution2D(iImage, iKernel)` izvaja diskretno 2D konvolucijo slike s poljubnim konvolucijskim jedrom z uporabo štirih gnezdnih zank. Najprej inicializira izhodno sliko enake velikosti kot vhodna, nato za vsak piksel (x, y) v izhodni sliki sešteje produkt vrednosti okoliških slikovnih točk (tx, ty) in ustreznih vrednosti jedra pri indeksih (u, v) , pri čemer upošteva robne pogoje s preverjanjem veljavnih indeksov. Funkcija uporablja uteženo povprečje z zamikom jedra tako, da je center jedra nad pikslom za katerega računamo izhodno vrednost. Za celoštevilske podatkovne tipe funkcija rezultat omeji na dovoljeno območje vrednosti z uporabo funkcije `np.clip`, kar prepreči prekoračitev območja zaloge vrednosti. Rešitev lahko zapišemo:

```
1 def discreteConvolution2D( iImage, iKernel ):
2     """Diskretna 2D konvolucija slike s poljubnim jedrom"""
3     # pretvori vhodne spremenljivke v np polje in
```

```

4      # inicializiraj izhodno np polje
5      iImage = np.asarray( iImage )
6      iKernel = np.asarray( iKernel )
7      oImage = np.zeros_like( iImage, dtype='float' )
8      # preberi velikost slike in jedra
9      dy, dx = iImage.shape
10     dv, du = iKernel.shape
11     # izracunaj konvolucijo
12     for y in range( dy ):
13         for x in range( dx ):
14             for v in range( dv ):
15                 for u in range( du ):
16                     tx = int(x - u + np.floor(du/2))
17                     ty = int(y - v + np.floor(dv/2))
18                     if tx>=0 and tx<dx and ty>=0 and ty<dy:
19                         oImage[y, x] = oImage[y, x] + \
20                             float(iImage[ty, tx]) * float(iKernel[v, u])
21     if iImage.dtype.kind in ('u','i'):
22         oImage = np.clip(oImage,
23                             np.iinfo(iImage.dtype).min,np.iinfo(iImage.dtype).max)
23     return np.array( oImage, dtype=iImage.dtype )

```

Konvolucija je ključna operacija v obdelavi slik, ki se uporablja za filtracijo, zamegljevanje, izostritev in zaznavanje robov. Prikazana implementacija v Python-u zaradi gnezdenja ni učinkovita; naprednejše implementacije izkoriščajo tako strojno opremo kot nižjenivojski jezik C ali C++. V nalogi boste preverili učinkovitejšo implementacijo `convolve()` v Python knjižnici `scipy.ndimage`. Preizkus funkcije `discreteConvolution2D(iImage, iKernel)` naredimo na naslednji način:

```

1      #box filter
2      k = 11 # 5, 3
3      iKernel = np.ones([k, k])
4      iKernel = iKernel / np.sum(iKernel)
5      oImageG = discreteConvolution2D( iImageG, iKernel )
6      rvlb.showImage( oImageG, f'k={k}' )

```

Rezultatirajoči slike za $k = 5$ in $k = 11$ sta prikazani na sliki 5.2. Za večje jedro lahko opazimo temnenje roba slike, kar je posledica predpostavke, da so intenzitete izven slike privzeto vrednosti 0. To popačenje lahko odpravimo z ustrezno razširitvijo slike (glej nalogo).

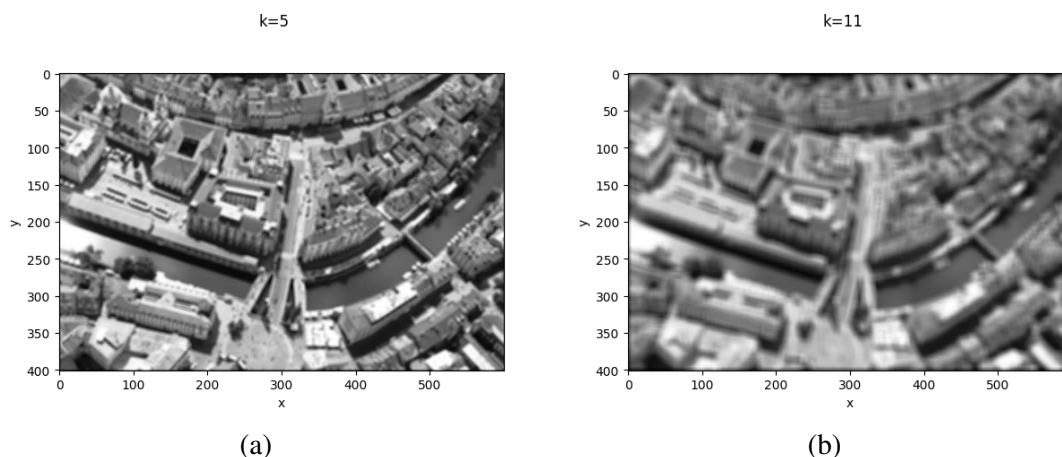
Vaja 5.2 Napišite funkcijo za interpolacijo ničtega reda, ki vzorči vhodno 2D sivinsko sliko `iImage`:

```

1      def interpolate0Image2D( iImage, iCoorX, iCoorY ):
2          # Tu napišite python kodo funkcije
3          return oImage

```

kjer so `iCoorX` in `iCoorY` vzorčne koordinate v prostoru vhodne slike `iImage`, v katerih želimo izračunati sivinsko vrednost. Izračunane sivinske vrednosti predstavljajo sivinske vrednosti izhodne slike `oImage`. Interpolacija ničtega reda priredi sivinsko vrednost v točki (x,y) enako sivinski vrednosti najbližje točke na diskretni vzorčni mreži. Pri iskanju najbližje točke v diskretni mreži si pomagajte z Python funkcijo `numpy.floor()`, za definicijo vzorčnih točk pa



Slika 5.2: Rezultat diskretne konvolucije z jedrom za lokalno aritmetično povprečenje (ang. *box filter*) z dimenzijami (a) 5×5 ($k = 5$) in (b) 11×11 ($k = 11$).

uporabite funkcijo `numpy.meshgrid(..., indexing='xy')`. Preizkusite delovanje funkcije na podsliki sivinske slike med ogliščema (260,210) in (360,280) tako, da podsliko prevzorčite s trikrat bolj gosto vzorčno mrežo, kot je mreža originalne sivinske slike. ■

Funkcija `interpolate0Image2D(iImage, iCoorX, iCoorY)` izvaja interpolacijo ničtega reda (najbližjega sosedu) za 2D sliko tako, da najprej pretvori vhodne koordinate v celoštevilске indekse z zaokrožanjem navzdol (`np.floor`), nato za vsako koordinato (x, y) preveri, ali leži znotraj meja slike, in priredi vrednost intenzitet najbližjega piksla. Ključne značilnosti vključujejo: avtomatsko ustvarjanje pravokotne mreže koordinat iz 1D razpona horizontalnih in vertikalnih indeksov z `np.meshgrid`, preverjanje veljavnosti indeksov za robustno obvladovanje robnih primerov, ter ohranjanje podatkovnega tipa originalne slike. Rešitev lahko zapišemo:

```

1 def interpolate0Image2D( iImage, iCoorX, iCoorY ):
2     """Funkcija za interpolacijo nitega reda"""
3     # pretvori vhodne spremenljivke v np polje
4     iImage = np.asarray( iImage )
5     iCoorX = np.asarray( iCoorX, dtype='float' )
6     iCoorY = np.asarray( iCoorY, dtype='float' )
7     # preberi velikost slike in jedra
8     dy, dx = iImage.shape
9     # ustvari 2d polje koordinat iz 1d razpona vhodnih koordinat (!!!)
10    if np.size(iCoorX) != np.size(iCoorY):
11        print('Število X in Y koordinat je različno!')
12        iCoorX, iCoorY = np.meshgrid(iCoorX, iCoorY, indexing='xy')
13    # zaokroži na najbližjo celostevilsko vrednost (predstavlja indeks!)
14    oShape = iCoorX.shape
15    iCoorX = np.floor(iCoorX.flatten()).astype('int')
16    iCoorY = np.floor(iCoorY.flatten()).astype('int')
17    # ustvari izhodno polje
18    oImage = np.zeros(iCoorX.shape, dtype=iImage.dtype )
19    print(iCoorX.shape)
20    print(iCoorY.shape)
21    # priredi vrednosti
22    for idx in range(oImage.size):
23        tx = iCoorX[idx]
24        ty = iCoorY[idx]
```



```

25     if tx>=0 and tx<dx and ty>=0 and ty<dy:
26         oImage[idx] = iImage[ty, tx]
27     # vrni izhodno sliko
28     return np.reshape( oImage, oShape )

```

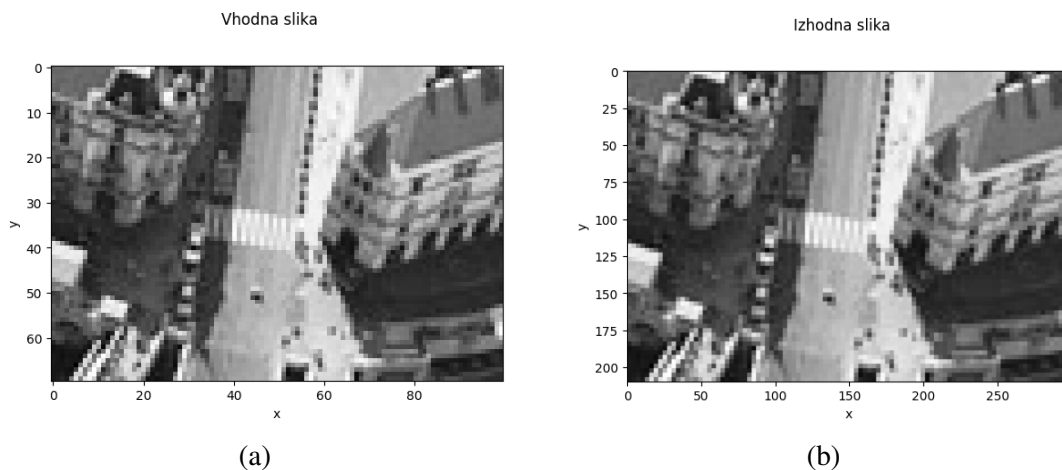
Interpolacija ničtega reda iz gladkih poševnih robov ustvari nazobčane robove. Po drugi strani je hitra, primerna za aplikacije, kjer je hitrost pomembnejša od gladkosti rezultata. Preizkus funkcije naredimo na naslednji način:

```

1 iImageGs = iImageG[210:280,260:360]
2 rvlb.showImage( iImageGs, 'Vhodna slika' )
3 # določi diskretno mrežo za interpolacijo
4 dy, dx = iImageGs.shape
5 iCoorX, iCoorY = np.meshgrid(np.arange(0,dx,1/3.0), \
6     np.arange(0,dy,1/3.0), indexing='xy')
7 # izvedi interpolacijo ničtega reda
8 oImageG = interpolate0Image2D( iImageGs, iCoorX, iCoorY )
9 rvlb.showImage( oImageG, 'Izhodna slika' )

```

Rezultat prikazuje slika 5.3.



Slika 5.3: Rezultat interpolacije ničtega reda (a) originalne podslike za (b) s korakom 1/3 piksla.

Vaja 5.3 Napišite funkcijo za interpolacijo prvega reda, ki vzorči vhodno 2D sivinsko sliko `iImage`:

```

1 def interpolate1Image2D( iImage, iCoorX, iCoorY ):
2     # Tu napišite python kodo funkcije
3     return oImage

```

kjer so `iCoorX` in `iCoorY` vzorčne koordinate v prostoru vhodne slike `iImage`, v katerih želimo izračunati sivinsko vrednost. Izračunane sivinske vrednosti predstavljajo sivinske vrednosti izhodne slike `oImage`. Interpolacija prvega reda priredi sivinsko vrednost v točki (x,y) enako linearno uteženi vsoti sivinskih vrednosti sosednjih štirih točk na diskretni vzorčni mreži (glej opis bilinearne interpolacije). Pri iskanju izhodiščne leve gornje točke v diskretni mreži si

pomagajte s funkcijo `numpy.floor()`. Preizkusite delovanje funkcije na podsliki sivinske slike med ogliščema (260,210) in (360,280) tako, da podsliko prevzorčite s trikrat bolj gosto vzorčno mrežo, kot je mreža originalne sivinske slike. ■

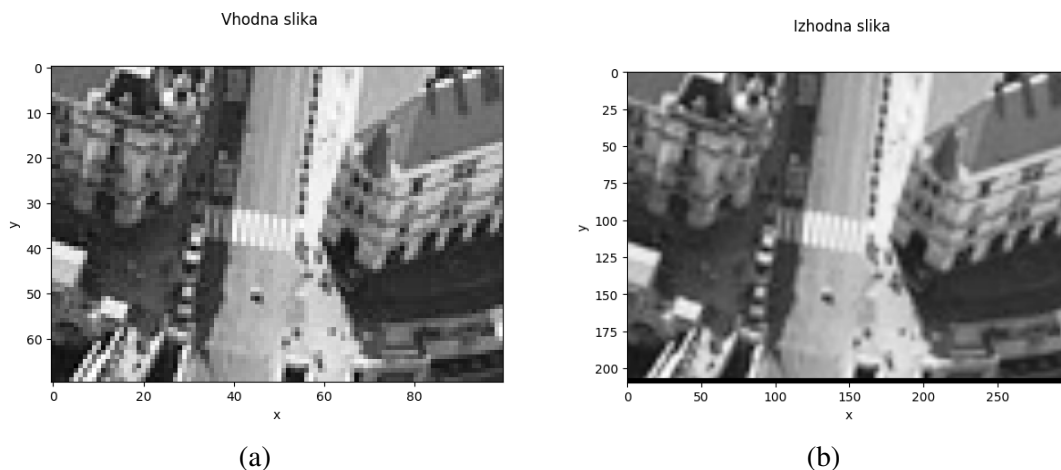
Funkcija `interpolate1Image2D(iImage, iCoorX, iCoorY)` izvaja bilinearno interpolacijo (prvega reda) 2D slike, kjer vrednost vsakega interpoliranega piksla izračuna kot uteženo povprečje štirih najbližjih sosedov v originalni sliki. Rešitev lahko zapišemo:

```

1 def interpolate1Image2D( iImage, iCoorX, iCoorY ):
2     """Funkcija za interpolacijo prvega reda"""
3     # pretvori vhodne spremenljivke v np polje
4     iImage = np.asarray( iImage )
5     iCoorX = np.asarray( iCoorX, dtype='float' )
6     iCoorY = np.asarray( iCoorY, dtype='float' )
7     # preberi velikost slike in jedra
8     dy, dx = iImage.shape
9     # ustvari 2d polje koordinat iz 1d vhodnih koordinat (!!!)
10    if np.size(iCoorX) != np.size(iCoorY):
11        print('Število X in Y koordinat je različno!')
12        iCoorX, iCoorY = np.meshgrid(iCoorX, iCoorY, indexing='xy')
13    # pretvori v linearno polje
14    oShape = iCoorX.shape
15    iCoorX = iCoorX.flatten()
16    iCoorY = iCoorY.flatten()
17    # ustvari izhodno polje, pretvori v linearno polje
18    oImage = np.zeros( iCoorX.shape , dtype='float' )
19    # priredi vrednosti
20    for idx in range(oImage.size):
21        lx = int(np.floor(iCoorX[idx]))
22        ly = int(np.floor(iCoorY[idx]))
23        sx = iCoorX[idx] - lx
24        sy = iCoorY[idx] - ly
25        if lx>=0 and lx<(dx-1) and ly>=0 and ly<(dy-1):
26            # izracunaj utezi
27            a = (1 - sx) * (1 - sy)
28            b = sx * (1 - sy)
29            c = (1 - sx) * sy
30            d = sx * sy
31            # izracunaj izhodno vrednost
32            oImage[idx] = a * iImage[ly, lx] + \
33                        b * iImage[ly, lx+1] + \
34                        c * iImage[ly+1, lx] + \
35                        d * iImage[ly+1, lx+1]
36    if iImage.dtype.kind in ('u','i'):
37        oImage[oImage<np.iinfo(iImage.dtype).min] = np.iinfo(iImage.dtype).min
38        oImage[oImage>np.iinfo(iImage.dtype).max] = np.iinfo(iImage.dtype).max
39    return np.array( np.reshape( oImage, oShape ), dtype=iImage.dtype )

```

Algoritem najprej določi osnovne koordinate (l_x, l_y) z zaokrožanjem navzdol in relativne pozicije (s_x, s_y) znotraj osnovne celice, nato izračuna uteži (a, b, c, d) za vsakega od štirih sosednjih pikslo v in izvede uteženo vsoto. Za celoštevilske podatkovne tipe funkcija zagotovi pravilno območje vrednosti z uporabo `np.clip`. Bilinearna interpolacija omogoča bolj gladke rezultate kot metoda najbližjega sosesa, vendar je računsko zahtevnejša. Preizkus funkcije naredimo na isti vhodni sliki kot pri vaji 5.3, le da zamenjamo funkciji za interpolacijo. Rezultat prikazuje slika 5.4.



Slika 5.4: Rezultat interpolacije prvega reda (a) originalne podslike za (b) s korakom 1/3 piksla. Primerjava s sliko 5.3b (interpolacija 0. reda) kaže na večjo *gladkost* robov.

Vaja 5.4 Napišite funkcijo za piramidno decimacijo sivinskih slik:

```
1 def decimateImage2D( iImage, iLevel ):
2     # Tu napišite python kodo funkcije
3     return oImage
```

kjer je *iImage* vhodna sivinska slika, *iLevel* pa število zaporednih decimacij vhodne slike s faktorjem dva. Pred vsako decimacijo filtrirajte sliko z nizkoprepustnim sitom velikosti 3×3 (glej opis piramidne decimacije) s funkcijo `discreteConvolution2D()`. Funkcija vrne decimirano sivinsko sliko *oImage*. Preizkusite delovanje funkcije na sivinski sliki za vrednosti *iLevel* = 1, 2, 3. ■

Funkcija `decimateImage2D(iImage, iLevel)` izvaja piramidno decimacijo slike z uporabo rekurzivnega pristopa. Najprej uporabi Gaussovo glajenje s 3×3 jedrom za preprečevanje preslikovanja višjih frekvenc v nižje (ang. *aliasing*), nato sliko decimira z vzorčenjem vsakega drugega piksla v obeh oseh slike. Postopek se rekurzivno ponavlja za določeno število nivojev (*iLevel*), kar ustvarja slikovno piramido z vedno manjšo ločljivostjo. Rešitev lahko zapišemo:

```
1 def decimateImage2D( iImage, iLevel ):
2     """Funkcija za piramidno decimacijo"""
3     print('Decimacija pri iLevel = ', iLevel)
4     # pretvori vhodne spremenljivke v np polje
5     iImage = np.asarray( iImage )
6     iImageType = iImage.dtype
7     # gaussovo jedro za glajenje
8     iKernel = np.array( ((1/16,1/8,1/16),(1/8,1/4,1/8),(1/16,1/8,1/16)) )
9     # glajenje slike pred decimacijo
10    # iImage = discreteConvolution2D( iImage, iKernel )
11    # hitrejša verzija glajenja
12    import scipy.ndimage as ni
13    iImage = ni.convolve( iImage, iKernel, mode='nearest' )
14    # decimacija s faktorjem 2
15    iImage = iImage[::2,::2]
16    # vrni sliko oz. nadaljuj po piramidi
17    if iLevel <= 1:
```

```

18     return np.array( iImage, dtype=iImageType )
19     else:
20     return decimateImage2D( iImage, iLevel-1 )

```

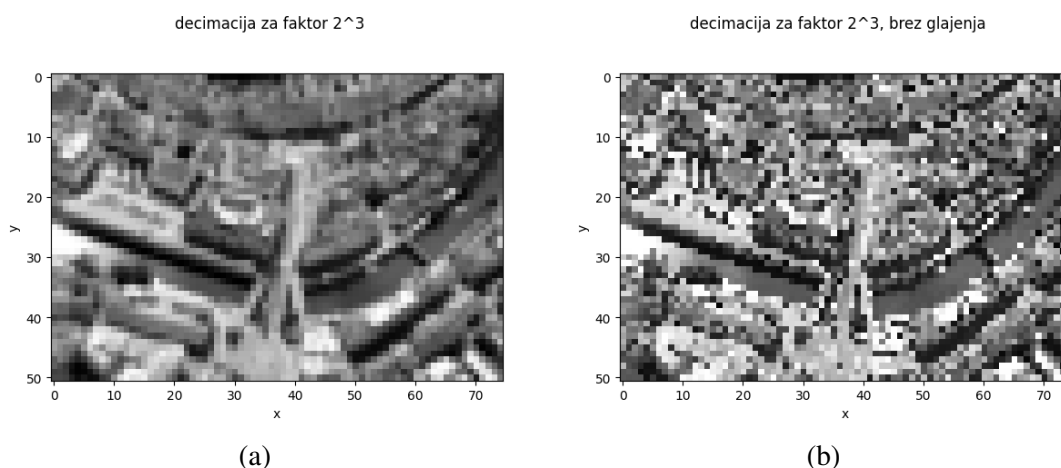
Za hitrejše izvajanje uporablja optimizirano funkcijo za konvolucijo iz knjižnice SciPy. Vsak nivo zmanjša ločljivost slike za faktor 2, pri čemer ohranja podatkovni tip originalne slike. Rekurzija se ustavi, ko doseže osnovni primer ($iLevel \leq 1$). Funkcijo smo preverili z decimacije originalne slike 5.1 z faktor $8\times$ ($iLevel = 3$) na naslednji način:

```

1 # zaženi funkcijo
2 oImageG = decimateImage2D( iImageG, 3 )
3 # nariši rezultat
4 rvlb.showImage( oImageG, 'decimacija za faktor 2^3' )

```

Preverimo lahko tudi rezultat brez uporabe glajenje (*zakomentiramo ustrezno vrstico funkcije*). Rezultat prikazuje slika 5.5, kjer se v primer neuporabe filtriranja v izhodni sliki pojavijo novi robovi, kot posledica preslikovanja višjih frekvenc v nižje.



Slika 5.5: Rezultat decimacije originalne slike 5.1 za faktor $8\times$ ($= 2^3$) (a) z Gaussovim filtriranjem in (b) brez filtriranja.

5.5 Naloge in vprašanja

Poročilo naloge pripravite v obliki Python skripte, ki naj izvede zahtevane izračune oz. klice funkcij in izriše pripadajoče slike oz. grafe. Vaša dolžnost je, da v skripti uvozite ustrezne Python knjižnice in dodate morebitne podporne skripte. Skripta se mora izvesti in poustvariti rezultate, kot se zahtevajo v nalogah. Na vprašanja odgovorite v obliki komentarja, naprimer *# Naloga X.2: To je moj odgovor..*

Naloga 5.1 V funkciji diskretne 2D konvolucije smo privzeli, da so sivinske vrednosti točk zunaj slike enake 0. Pri uporabi funkcije z večjimi konvolucijskimi jedri zaradi tega na izhodnih slikah dobimo popačen rob. Eden izmed načinov za odpravo teh popačenj je ta, da vhodno sliko v vse smeri razširimo z robom, ki je enak polovici širine konvolucijskega jedra ($U/2, V/2$). Sivinske vrednosti v razširjenem delu slike nadomestimo s sivinskimi vrednostmi, ki so enake sivinskim vrednostim najbližjih slikovnih elementov v originalni vhodni sliki.

Razširite funkcijo `discreteConvolution2D()` tako, da bo imela dodatni vhodni parameter `iPadImage` z vrednostjo `'zeros'` ali `'nearest'`. Parameter `iPadImage` naj definira način razširitve slike pred izračunom diskretne 2D konvolucije, in sicer za `iPadImage = 'zeros'` privzemite, da so sivinske vrednosti točk zunaj slike enake 0; za `iPadImage = 'nearest'` pa naj bodo enake sivinskim vrednostim najbližjih slikovnih elementov v originalni vhodni sliki. Prikažite delovanje razširjene funkcije na sivinski sliki z jedrom `iKernel = numpy.ones([11,11])/121`.

Naloga 5.2 Uporabite funkcijo `convolve()` v Python knjižnici `scipy.ndimage` za izračun 2D diskretne konvolucije na sivinski sliki z jedrom `iKernel = numpy.ones([11,11])/121`. Primerjajte rezultate in čas izračuna (v milisekundah). Čas izračunajte z uporabo funkcije `clock()` v knjižnici `time` in ga nato ustrezno pretvorite.

Naloga 5.3 V funkciji diskretne 2D konvolucije napisani na vajah smo privzeli, da so sivinske vrednosti točk zunaj slike enake 0. Pri uporabi funkcije z večjimi konvolucijskimi jedri zaradi tega na izhodnih slikah dobimo popačen rob. Obstajajo različni načini za odpravo teh popačenj. Opišite in analizirajte vse možne načine, ki jih omogoča funkcija `convolve()` v Python knjižnici `scipy.ndimage` (parameter `mode`), ter prikažite rezultate. Vizualno ocenite kako se razlikujejo rezultati različnih načinov pri manjših in večjih velikostih jedra za povprečenje (npr. 5 in 21).

Naloga 5.4 Napišite funkcijo `discreteConvolutionColorImage`, ki bo omogočala konvolucijo barvnih slik s poljubnim jedrom. Preizkusite delovanje na barvni sliki z jedrom `iKernel = numpy.ones([11,11])/121` in prikažite sliko.

```
1 def discreteConvolutionColorImage( iImage, iKernel ):
2     # Tu napišite python kodo funkcije
3     return oImage
```

Naloga 5.5 Napišite funkcijo za izračun konvolucijskega jedra v obliki 2D simetrične Gaussove funkcije:

```
1 def discreteGaussian2D( iSigma ):
2     # Tu napišite python kodo funkcije
3     return oKernel
```

kjer je `iSigma` standardna deviacija σ v 2D simetrični Gaussovi funkciji, ki je definirana kot:

$$k(u,v) = (2\pi)^{-1} \sigma^{-2} \exp^{-(u^2+v^2)/2\sigma^2}. \quad (5.2)$$

Velikost konvolucijskega jedra $U \times V$ nastavite glede na vrednost `iSigma` kot $U, V = 2 \cdot \lceil 3 \cdot \sigma \rceil + 1$, kjer tako pridobljene vrednosti U in V zaokrožite s funkcijo `numpy.ceil`, vrednost $k(0,0)$ pa naj bo v središčnem elementu izhodne 2D matrike `oKernel`. Zagotovite, da bo vsota vseh elementov `oKernel` enaka 1. Določite konvolucijski jedri za $\sigma = 0.1$ in 3. Z jedroma izvedite diskretno 2D konvolucijo na barvni sliki in ju prikažite. Obrazložite vpliv vrednosti σ .

Naloga 5.6 Ostrenje slike je analogno prostorskemu odvajanju sivinskih vrednosti. Ostrenje slik z drugim odvodom izvedemo z uporabo Laplaceovega jedra tako, da vhodni sliki $g(x,y)$ odštejemo uteženo sliko drugega odvoda slike:

Laplaceovo jedro

$$f(x,y) = g(x,y) - c \Delta g(x,y),$$

0	1	0
1	-4	1
0	1	0

kjer konstanta c določa stopnjo ostrenja. Po odštevanju oz. prištevanju slike drugega odvoda $\Delta g(x,y)$ zagotovite, da bodo vrednosti 8-bitne sivinske slike na območju od $[0, 255]$, pri čemer vrednosti manjše od 0 oz. vrednosti večje od 255 ustrezno zaokrožite. Izostrite sivinsko sliko z vrednostmi $c = 0, 0.5, 1, 2$ in slike prikažite. Preizkusite različne vrednosti c in obrazložite njihov vpliv na rezultirajočo sivinsko sliko. Preizkusite ostrenje še na barvni sliki.

Naloga 5.7 Uporabite funkcijo `interp()` v Python knjižnici `scipy.interpolate` za izračun 2D interpolacije ničtega in prvega reda, podobno kot pri nalogah 2 in 3 iz vaj. Primerjajte rezultate in čas izračuna.

Naloga 5.8 Napišite funkcijo `interpolateColorImage`, ki bo omogočala uporabo interpolacije ničtega (`method=nearest`) in prvega (`method=linear`) reda na barvnih slikah. Preizkusite delovanje funkcije tako, da z ničtim in prvim redom prevzorčite celotno barvno sliko z dvakrat bolj gosto vzorčno mrežo, kot je mreža originalne barvne slike.

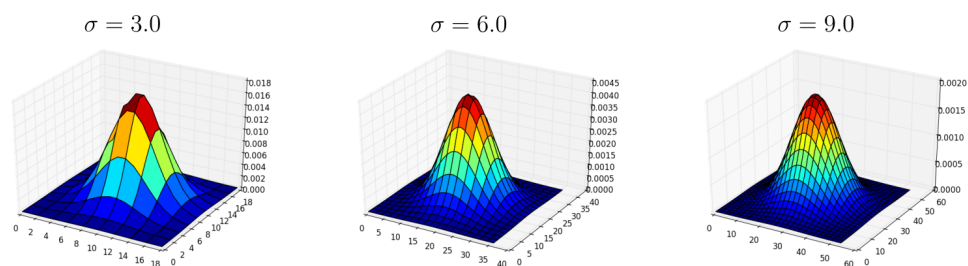
```
1 def interpolateColorImage( iImage, iCoorX, iCoorY, method ):
2     # Tu napišite python kodo funkcije
3     return oImage
```

Naloga 5.9 Interpolacijo slike lahko uporabljamo tudi za povečavo delov slike. Z interpolacijo prvega reda izvedite petkratno ($5\times$) povečavo pravokotnega področja barvne slike med ogliščema $(100, 50)$ in $(250, 200)$. Prikažite originalno in povečano pravokotno področje barvne slike.

Naloga 5.10 Prilagodite funkcijo za piramidno decimacijo tako, da bo omogočala decimiranje barvnih slik in preizkusite delovanje na barvni sliki. Pomanjšajte sliko za 2-krat in 4-krat.

Rezultati nekaterih nalog:

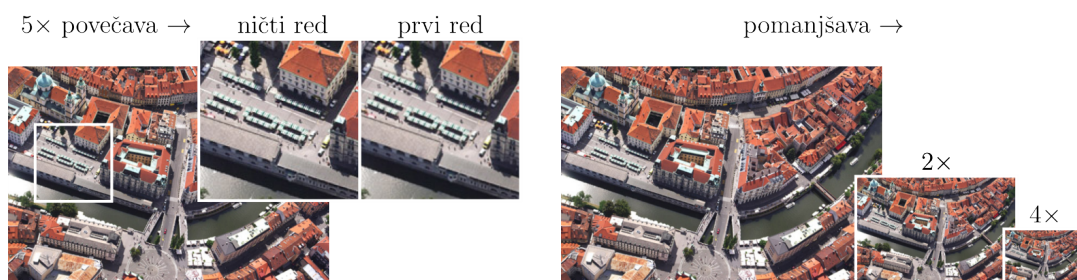
Gaussova funkcija pri različnih vrednostih parametra σ



Ostrenje slike z uporabo Laplaceovega jedra



Interpolacija in decimacija barvnih slik





6. Robustno iskanje objektov

Pri vaji bomo najprej obravnavali nizkonivojske postopke za iskanje oz. detekcijo oglišč in robov v 2D slikah, ki temeljijo na analizi 1. odvoda oz. gradienta sivinske slike. Oglišča so izraziti strukturni elementi slike oz. objektov na sliki in jih lahko uporabimo v aplikacijah kot so sledenje objektov v zaporednih posnetkih, določanje preslikave med različnimi pogledi enakega objekta, kot referenčne točke za geometrijske meritve, za kalibracijo optičnih sistemov itd. Detektirane robove pa lahko nadgradimo za zaznavanje kompleksnejših objektov v 2D slikah.

V nadaljevanju vaje bomo obravnavali Houghovo preslikavo, ki omogoča zaznavo parametričnih oblik na osnovi zaznanih robov v sliki in s tem detekcijo predvsem enostavnih umetnih objektov (npr. pravokotne, okrogle, eliptične strukture). Za zaznavanje naravnih objektov, kjer so robovi lahko poljubnih oblik, pomembne pa so tudi druge značilnice, kot so barva, tekstura in kontekst, pa bomo uporabili konvolucijske nevronske mreže.

Pri vaji boste sami implementirali funkcije za osnovne detektorje oglišč, robov in Houghovo preslikavo, kasneje pa preizkusili še obstoječe implementacije v knjižnici OpenCV in primerjali rezultate. OpenCV je knjižnica programskih funkcij, ki se izvajajo v realnem času, namenjenim potrebam računalniškega vida. Knjižnica vsebuje osnovna orodja za obdelavo slik, torej tudi tista za robustno iskanje 2D objektov, s katerimi se bomo seznanili pri vaji. Za preizkus nevronske mreže boste uporabili implementacije in prednaučene modele v tej knjižnici. OpenCV knjižnico naložite tako, da v ukazno vrstico terminala (ang. *command prompt*) vpišete:

```
pip install opencv-python opencv-contrib-python
```

V sledečih podpoglavjih so podane izpeljave algoritmov, ki so temelj za kasnejše programske implementacije.

6.1 Gradient sivinske slike

Oglišče je definirano kot presečišče dveh robov, pri čemer rob v sliki v splošnem lahko opišemo kot veliko lokalno spremembo sivinske vrednosti. Robove lahko detektiramo na podlagi odvodov

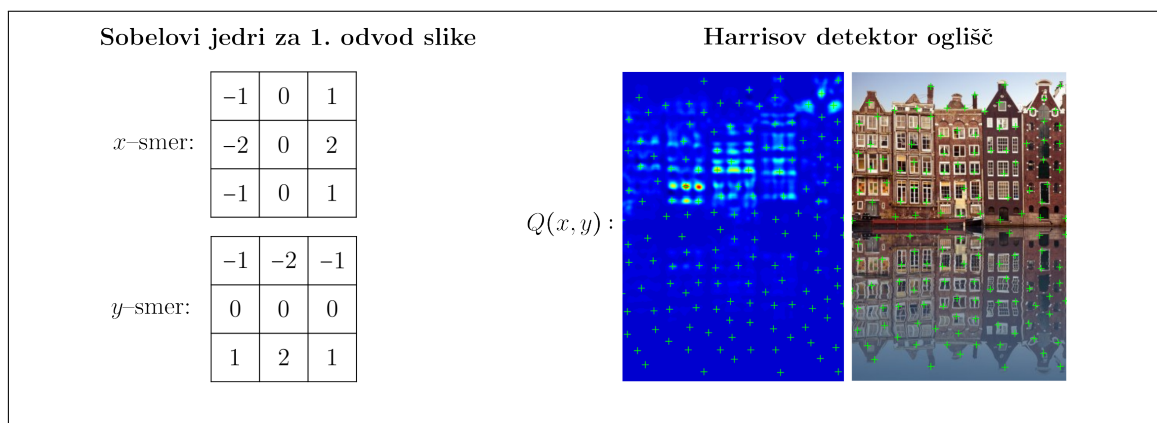
oz. gradientov slike. Prvi odvod za poljubno 2D funkcijo $f(x,y)$ zapišemo v obliki vektorja:

$$\nabla f(x,y) = \vec{g}(x,y) = \begin{bmatrix} g_x(x,y) \\ g_y(x,y) \end{bmatrix} = \begin{bmatrix} \partial f(x,y)/\partial x \\ \partial f(x,y)/\partial y \end{bmatrix}. \quad (6.1)$$

Vektorska slika gradienta $\vec{g}(x,y)$ v vsaki točki (x,y) vsebuje vektor, ki s koordinatno osjo x oklepa kot α in ima dolžino G :

$$\alpha(x,y) = \arctan \frac{g_y(x,y)}{g_x(x,y)}, \quad G(x,y) = \sqrt{g_x^2(x,y) + g_y^2(x,y)}. \quad (6.2)$$

Gradient $\vec{g}(x,y)$ kaže v smeri največje spremembe funkcije $f(x,y)$, torej pravokotno na dani rob v sliki. Komponenti gradienta $g_x(x,y)$ in $g_y(x,y)$, ki predstavljata parcialna odvoda, na digitalni sliki izračunamo z digitalnimi filtri, naprimer s **Sobelovim jedrom**. Vektorsko sliko $\vec{g}(x,y)$ odvodov dobimo z dvema 2D diskretnima konvolucijama med vhodno sivinsko sliko in Sobelovima jedroma (slika 6.1 *levo*).



Slika 6.1: Sobel jedri (*levo*) in odziv Harrisovega detektorja oglišč ter zaznana oglišča (*desno*).

Zvezno obliko gradientnega operatorja, ki omogoča zasnovo poljubno velikega digitalnega filtra z nastavlljivo stopnjo glajenja dobimo z odvajanjem Gaussove funkcije $\mathcal{N}(x,y | \sigma)$:

$$\nabla \mathcal{N}(x,y | \sigma) = \nabla \exp \left(-\frac{x^2 + y^2}{2\sigma^2} \right) = \begin{bmatrix} -x \\ -y \end{bmatrix} \frac{1}{\sigma^2} \exp \left(-\frac{x^2 + y^2}{2\sigma^2} \right). \quad (6.3)$$

Stopnjo glajenja poljubno izberemo s parametrom standardne deviacije σ .

6.2 Harrisov detektor oglišč

Robovi v slikah imajo velik gradient v smeri pravokotno na rob, v ogliščih pa je gradient slike velik v več smereh, zato večina detektorjev oglišč temelji na primerjavi velikosti komponent gradienta v x in y smeri slike. V praksi se pogosto uporablja **Harrisov detektor oglišč**, pri katerem za vsako točko (x,y) izračunamo lokalno strukturno matriko M :

$$M(x,y) = \begin{bmatrix} A(x,y) & C(x,y) \\ C(x,y) & B(x,y) \end{bmatrix} * N(x,y | \sigma) = \begin{bmatrix} g_x^2(x,y) & g_x(x,y)g_y(x,y) \\ g_x(x,y)g_y(x,y) & g_y^2(x,y) \end{bmatrix} * N(x,y | \sigma) \quad (6.4)$$

Komponente lokalne strukturne matrike M , tj. slike $A(x,y)$, $B(x,y)$ in $C(x,y)$ predstavljajo odvode slike, ki jih zgladimo z Gaussovim jedrom $N(x,y | \sigma)$ z uporabo diskretne 2D konvolucije (*).

Kriterijsko funkcijo za detekcijo oglišč v vsaki točki slike (x, y) določimo z analizo lastnih vrednosti λ_1 in λ_2 matrike M :

$$\lambda_{1,2} = \frac{\text{trace}(M)}{2} \pm \sqrt{\left(\frac{\text{trace}(M)}{2}\right)^2 - \det(M)}, \quad (6.5)$$

kjer sta $\text{trace}(\cdot)$ in $\det(\cdot)$ funkciji za sled in determinanto matrike M . Na področjih slike s konstantno sivinsko vrednostjo bo $M = 0$ in $\lambda_1 = \lambda_2 = 0$, pri enakomerno naraščujoči sivinski vrednosti v eni smeri pa bo $\lambda_1 > 0$ in $\lambda_2 = 0$. Lastni vrednosti λ_1 in λ_2 torej kodirata izrazitost roba, pripadajoča lastna vektorja pa smer roba. V ogliščih je rob izrazit v smeri obeh lastnih vektorjev, lastni vrednosti sta veliki $|\lambda_{1,2}| \gg 0$ in velja $|\lambda_1| \approx |\lambda_2|$, rob pa je tem bolj izrazit, čim manjša je razlika med lastnima vrednostima. Od tod sledi kriterijska funkcija za detekcijo oglišč:

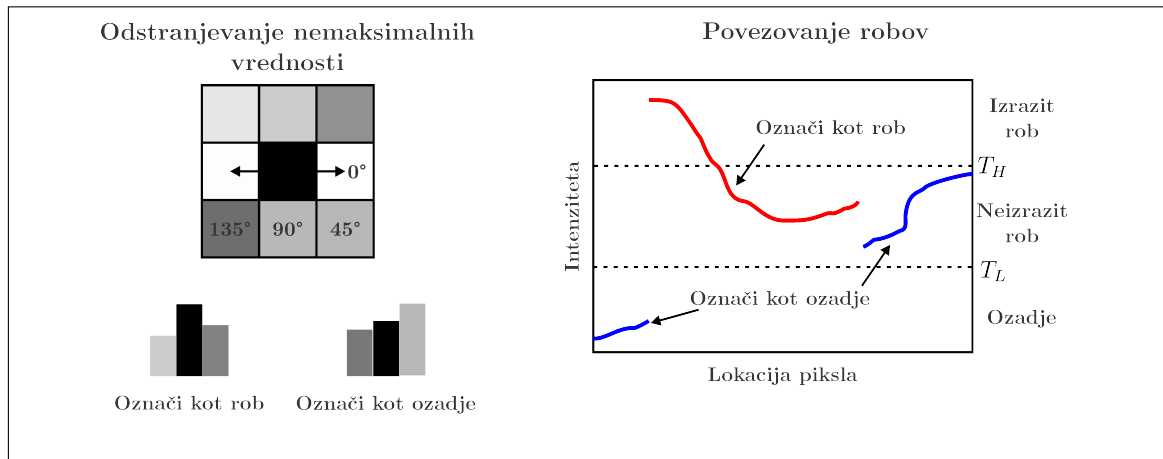
$$Q_H(x, y) = \lambda_1 \lambda_2 - \kappa (\lambda_1 + \lambda_2)^2 = \det(M) - \kappa (\text{trace}(M))^2. \quad (6.6)$$

Parameter κ določa občutljivost detektorja in ga izberemo na intervalu $[0, \frac{1}{4}]$. Točke roba detektiramo kot lokalne maksimume v $Q_H(x, y)$, nadalje pa detektirana oglišča prečistimo z upravljanjem $Q(x, y) > T_{min}$, kjer je T_{min} običajno na intervalu od 10^4 do 10^6 . Primer polja $Q(x, y)$ in zaznanih oglišč je prikazan na sliki 6.1 *desno*.

6.3 Cannyev detektor robov

Zaznavanje robov v slikah je eden izmed pomembnejših postopkov na področju obdelave in analize slik, **Cannyev detektor robov** pa je eden izmed najpogostejše uporabljenih tovrstnih postopkov. Algoritem Cannyevega detektorja lahko povzamemo s štirimi osnovnimi koraki:

1. glajenje slike z Gaussovim filtrom,
2. izračun slike velikosti in smeri gradienta,
3. odstranjevanje nemaksimalnih vrednosti,
4. dvojno upravljanje in povezovanje robov.



Slika 6.2: Odstranjevanje nemaksimalnih vrednosti (*levo*) in povezovanje robov (*desno*).

Za glajenje in določanje slik gradienta lahko uporabimo poljubni digitalni filter. V sliki gradientov dobimo okoli robov širša področja ojačitve, ki jih moramo zožiti oz. odstraniti, zato da ohranimo prave robove slike. Odstranjevanje nemaksimalnih vrednosti naredimo tako, da postavimo na nič vrednosti tistih slikovnih elementov, ki nimajo maksimalnih vrednosti v diskretni smeri gradienta ($0^\circ, 45^\circ, 90^\circ$ in 135°) oz. katerih sosednji slikovni elementi imajo v smeri gradienta

večje vrednosti (slika 6.2 *levo*). V zadnjem koraku postopka izvedemo še upravljanje slike robov in povezovanje robnih točk. Upravljanje izvedemo z dvema pragoma. Z zgornjim pragom T_H določimo izrazite robne točke $r_H(x, y)$, ki jih bomo neposredno zadržali. To so tiste točke, ki imajo magnitudo gradienta $G(x, y) > T_H$. S spodnjim pragom T_L pa določimo neizrazite robne točke $r_L(x, y)$, ki jih bomo obdržali za postopek povezovanja robov. Povezovanje izrazitih robov $r_H(x, y)$ izvedemo tako, da vsako sosednjo točko, ki pripada $r_L(x, y)$ ($T_L < G(x, y) \leq T_H$) vključimo v množico končnih robnih točk (slika 6.2 *desno*).

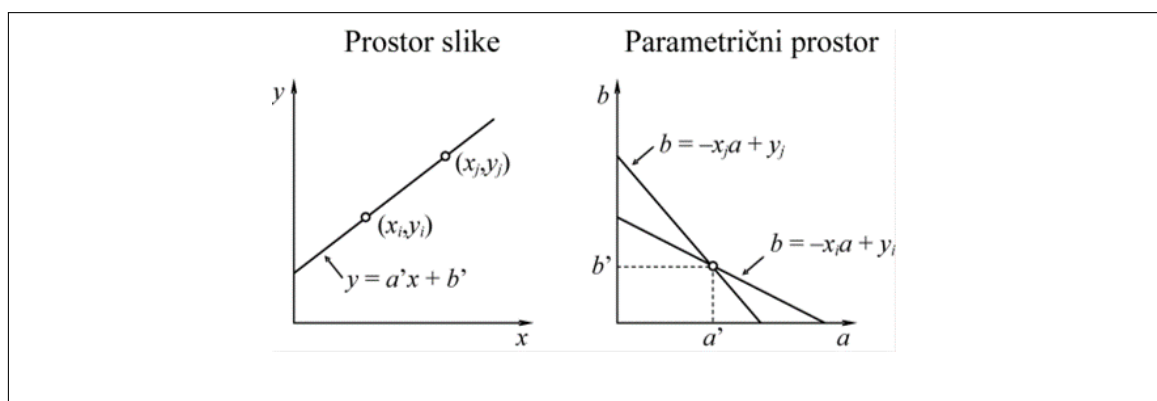
6.4 Houghov algoritem

Med postopke iskanje objektov na sliki spada tudi iskanje premic oz. črt. Učinkovit postopek iskanja premic na sliki temelji na **Houghovi transformaciji**. Predpostavimo, da imamo binarno sliko robov $r(x, y)$. Skozi poljubno izbrano robno točko (x_i, y_i) lahko potegnemo poljubno število premic oblike:

$$y_i = ax_i + b. \quad (6.7)$$

Isto enačbo lahko zapišemo v prostoru parametrov (a, b) , kjer sta koordinati robne točke x_i in y_i parametra premice:

$$b = -x_i a + y_i. \quad (6.8)$$



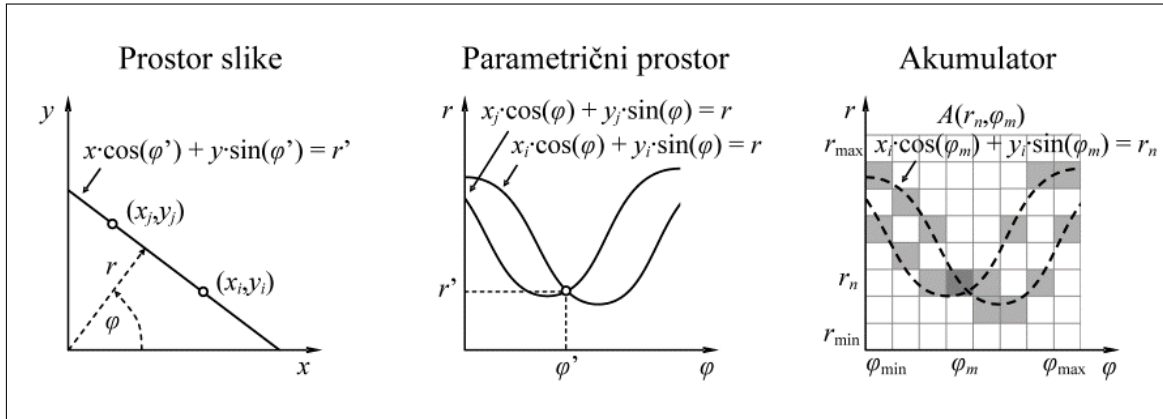
Slika 6.3: Preslikava točke roba iz prostora slike v parametrični prostor za eksplisitno obliko zapisa premice ($y = ax + b$).

Za neko drugo robno točko (x_j, y_j) dobimo v prostoru parametrov (a, b) še eno premico, ki se v neki točki (a', b') seka s premico, ki pripada (x_i, y_i) . Parametra (a', b') določata enačbo premice v prostoru slike $r(x, y)$, ki gre skozi točki (x_i, y_i) in (x_j, y_j) . Če za vse robne točke v sliki vnesemo pripadajočo premico v parametrični prostor, potem lahko poiščemo premice oz. črte z detekcijo vrhov v parametričnem prostoru. Situacijo prikazuje slika 6.3.

Pri tovrstnem zapisu enačbe premice naletimo na problem pri navpičnih premicah, pri katerih gre parameter $a \rightarrow \text{inf}$, s tem pa tudi velikost parametričnega prostora (a, b) . Temu se lahko izognemo z zapisom premice v Hessejevi obliki oz. v polarnih koordinatah (r, φ) :

$$x \cos \varphi + y \sin \varphi = r \quad (6.9)$$

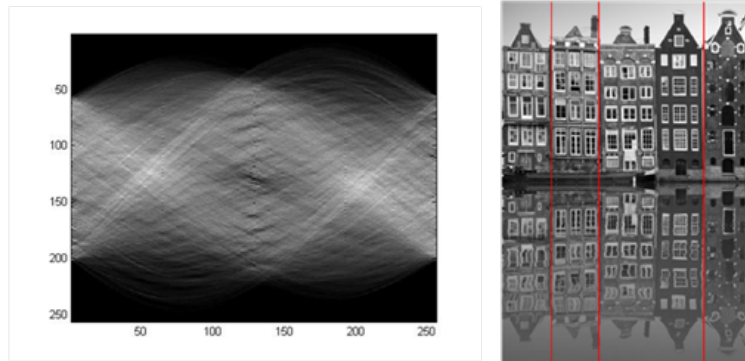
Za navpično premico ($\varphi = 0^\circ$) določa r presečišče z x osjo, za vodoravno premico ($\varphi = 90^\circ$) pa določa r presečišče premice z y osjo. V polarnem prostoru (r, φ) vsaka sinusna krivulja $x_i \cos \varphi + y_i \sin \varphi = r$ predstavlja množico premic, ki gredo v prostoru slike (x, y) skozi točko (x_i, y_i) .



Slika 6.4: Preslikava točke roba iz prostora slike v parametrični prostor za Hessejevo obliko zapisa premice ($x \cos \varphi + y \sin \varphi = r$).

Presečišče dveh ali večih sinusnih krivulj (r' , φ') določa parametra premice v prostoru (x, y) , kot prikazuje slika 6.4 *sredina*.

Prostor parametrov (r, φ) lahko enostavno diskretiziramo, in sicer $0 \leq \varphi < \pi$ ter $-r_{diag} \leq r \leq r_{diag}$ (slika 6.4 *desno*). Pri tem je $r_{diag} = \sqrt{M^2 + N^2}/2$ in M, N sta višina in širina slike $r(x, y)$, referenčna točka pa je center slike $(M/2, N/2)$. Diskretiziran prostor imenujemo akumulator $A(r_n, \varphi_m)$, ki ga najprej postavimo na 0, nato pa za vsako robno točko (x_i, y_i) in za vsako možno diskretno vrednost $0 \leq \varphi_m < \pi$ izračunamo ustrezno diskretno vrednost parametra r_n , nato pa vrednost akumulatorja v celici (r_n, φ_m) povečamo za 1. Premice, ki predstavljajo robne točke slike, določimo tako, da poiščemo tiste celice akumulatorja $A(r_n, \varphi_m)$, ki imajo dovolj velike in lokalno največje vrednosti. Primer rezultata je prikazan na sliki 6.5.



Slika 6.5: Akumulator po Houghovi preslikavi robov s Hessejevo obliko premice (*levo*) in premice, ki pripadajo zaznamim maksimumom v akumulatorju (*desno*).

6.5 Konvolucijske nevronske mreže

Konvolucijske nevronske mreže (CNN; ang. *Convolutional Neural Networks*) so specializirane arhitekture globokega učenja za analizo strukturiranih podatkov kot so digitalne slike. Njihova osnovna zasnova temelji na:

- **Konvolucijskih slojih:** avtomatično učijo hierarhije prostorskih značilnic z lokalnimi filtri na osnovi diskretne konvolucije,

- **Pooling slojih:** zmanjšujejo prostorsko ločljivost za povečanje invariantnosti na transformacije, na primer z uporabo sloja združevanja (ang. *pooling layer*),
- **Povezovalnih slojih:** klasične nevronske mreže, kot na primer večplastni perceptron, ki kombinirajo visokonivojske značilnice za končno razvrščanje.

Najbolj uveljavljene arhitekture so **Faster R-CNN** [15], **YOLO** (ang. *You Only Look Once*) [20], **SSD** (ang. *Single Shot MultiBox Detector*) [19], **RetinaNet** [18] in **EfficientDet** [21]. Sodobne arhitekture vsebujejo tudi napredne arhitekture in mehanizme, kot na primer mehanizem pozornosti za dinamično uteževanje pomembnih prostorskih lokacij, večnivojsko analizo značilnic in vizualne transformerje (ViT [12], DETR [16]) kot alternativne pristope.

Knjižnica OpenCV vsebuje podporo za več vnaprej naučenih globokih modelov prek modula `cv2.dnn`. Modeli se delijo glede na namen uporabe, na primer za **detekcijo objektov**:

- **MobileNet-SSD:** hitra detekcija objektov optimizirana za mobilne naprave (20 razredov),
- **YOLOv3/YOLOv4-tiny:** pospešene različice YOLO za realnočasovno detekcijo (80 razredov),
- **Faster R-CNN:** natančnejši model za detekcijo objektov (različni podatkovni nizi),
- **EfficientDet:** učinkovite arhitekture za večnamensko zaznavanje.

Ti modeli vrnejo oznako kategorije objekta (oz. razred objekta) in pripadajočo verjetnost. Za **semantično segmentacijo**, ki vsakemu pikslu v sliki pripiše pripadajočo oznako lahko uporabimo:

- **ENet:** lahka arhitektura za semantično segmentacijo,
- **FCN:** polno konvolucijske mreže za segmentacijo.

Za **razvrščanje slik** glede na kategorije v bazi ImageNet [17] so na voljo:

- **AlexNet/GoogleNet/ResNet:** klasične arhitekture za klasifikacijo,
- **SqueezeNet:** kompaktni model z visoko natančnostjo.

Na voljo so tudi **specifični modeli**:

- **OpenPose:** detekcija človeških poz,
- **Text Detection (EAST):** detekcija besedila v naravnih scenah,
- **Style Transfer:** umetniški prenos sloga med slikami.

Modeli so na voljo v več formatih:

- **TensorFlow** (.pb konfiguracija in uteži),
- **Caffe** (.prototxt in .caffemodel),
- **Darknet** (.cfg in .weights za YOLO),
- **ONNX** (.onnx format).

V splošnem je za uporabo modelov potrebno izvesti naslednje korake:

1. prenesti konfiguracijsko datoteko in prednaučene uteži,
2. naložiti model z `cv2.dnn.readNet()`,
3. pripraviti vhodne podatke z `cv2.dnn.blobFromImage()`,
4. izvesti napoved prek `net.forward()`.

Primer uporabe GoogLeNet mreže boste spoznali pri vaji.

6.6 Vaje z rešitvami

Pri vaji boste sami implementirali funkcije za Harrisov detektor oglišč in Cannyenv detektor robov, medtem ko boste uporabili funkcije v knjižnici OpenCV za detekcijo parametričnih objektov (premic, krogov) s Houghovim algoritmom. Vajo začnemo z izračunom gradientnega polja sivinske slike in njegovo upodobitvijo, ker kasneje na osnovi analize polja gradientov iščemo tako oglišča kot robove. Koda vaje je dostopna na [Git repozitoriju](#).

6.6.1 Gradient sivinske slike

Najprej uvozimo Python knjižnice za vajo:


```

1 import matplotlib.pyplot as plt
2 import scipy.ndimage as ni
3 import numpy as np
4 from rvlb import *
5 import cv2

```

Vaja 6.1 Napišite funkcijo za izračun 1. odvoda vhodne sivinske slike iImage:

```

1 def imageGradient( iImage ):
2     # Tu napišite kodo funkcije
3     return oGx, oGy

```

Funkcija vrne sliki parcialnih odvodov oGx in oGy, ki imata enako velikost kot vhodna slika iImage. Naložite barvno *RGB* sliko *slika1.jpg* v okolje Python in jo pretvorite v sivinsko sliko po enačbi $S = 0,299R + 0,587G + 0,114B$. Izračunajte prvi odvod sivinske slike in ga prikažite kot vektor v vsaki točki slike tako, da uporabite funkcijo `matplotlib.pyplot.quiver()`.

Rešitev lahko zapišemo z uporabo Sobelovega jedra in diskretne konvolucije:

```

1 def imageGradient( iImage ):
2     """Gradient slike s Sobelovim operatorjem"""
3     iImage = iImage.astype('float')
4     iSobel = np.array( ((-1,0,1),(-2,0,2),(-1,0,1)) )
5     oGx = ni.convolve( iImage, iSobel, mode='nearest' )
6     oGy = ni.convolve( iImage, np.transpose( iSobel ), mode='nearest' )
7     return oGx, oGy
8
9 # preizkus funkcije
10 iImage = loadImage( 'slika1.jpg' )
11 iImageG = colorToGray(iImage)
12 oGx, oGy = imageGradient(iImageG)
13
14 # prikaz slik
15 showImage(iImageG, 'Sivinska slika')
16 showImage(oGx, 'Gx')
17 showImage(oGy, 'Gy')
18
19 # prikaz vektorskega polja (decimirano s faktorjem 2, za hitrejši prikaz)
20 dy, dx = iImageG.shape
21 oCx, oCy = np.meshgrid(range(0,dx,2), range(0,dy,2), indexing='xy')
22 plt.quiver(oCx, oCy, oGx[::2,::2], -oGy[::2,::2], color='r')

```

Sivinsko različico vhodne barvne slike in rezultat izračuna gradienta v horizontalni (oGx) in vertikalni (oGy) smeri ter upodobitev gradientnega polja z sliko vektorjev prikazuje slika 6.6.

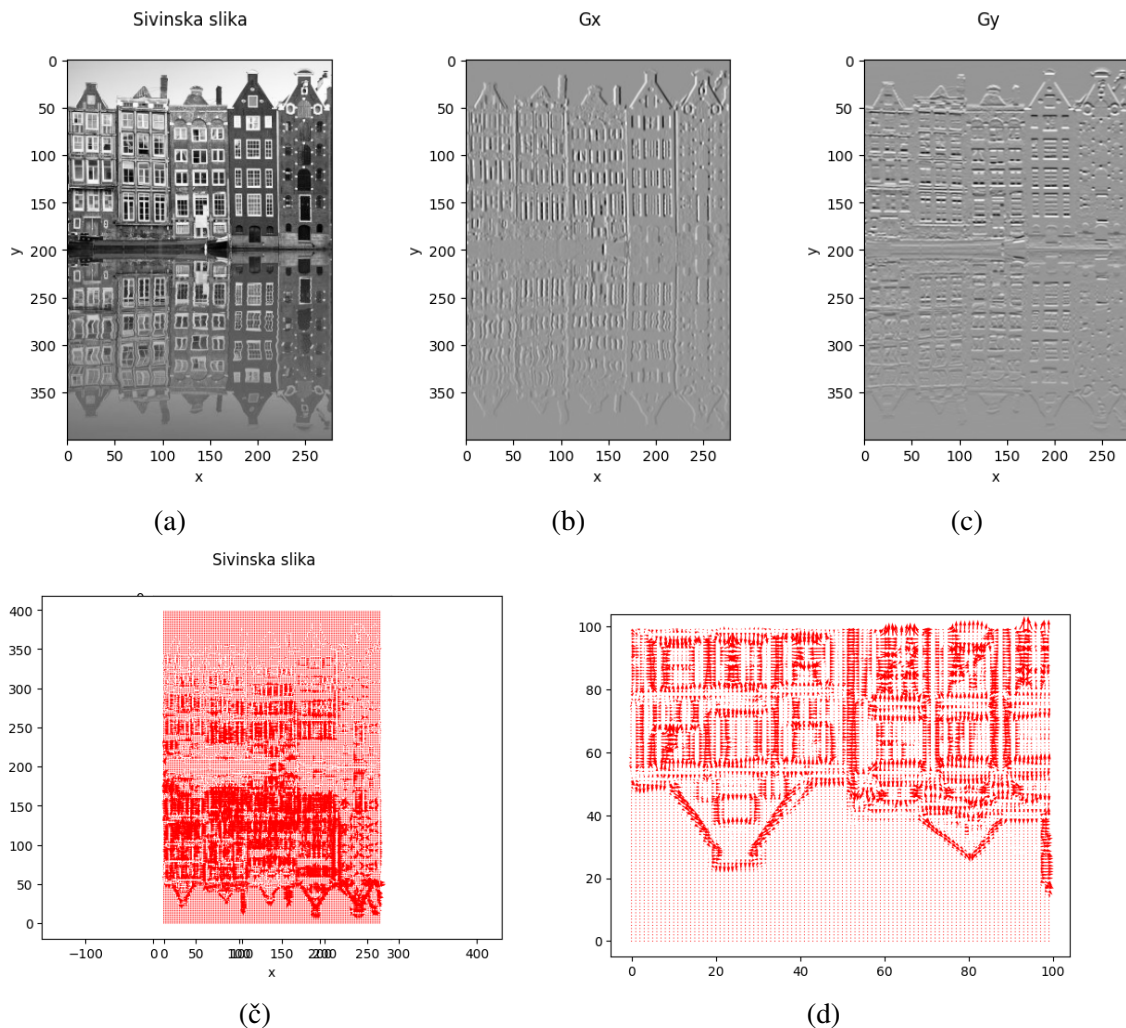
6.6.2 Harrisov detektor oglišč

Vaja 6.2 Napišite funkcijo za izračun odziva Harrisovega detektorja oglišč na vhodni sivinski sliki iImage:

```

1 def responseHarris( iImage, iKappa, iSigma ):

```



Slika 6.6: Prikaz (a) sivinske slike in pripadajoče (b) slike horizontalnega gradienta $g_x(x,y)$ in (c) vertikalnega gradienta $g_y(x,y)$; obe komponenti gradienta sta (č) upodobljeni kot vektorsko polje, za boljši vpogled je dana (d) povečava spodnjega levega kota velikosti 100×100 .

```
2 # Tu napišite kodo funkcije
3 return oQH
```

kjer je $iKappa$ občutljivost detektorja oglišč (κ), $iSigma$ pa standardna deviacija Gaussove funkcije za glajenje odziva. Funkcija vrne sliko oQH , ki predstavlja odziv detektorja oglišč. Preizkusite delovanje funkcije za različne vrednosti parametrov $iKappa$ in $iSigma$. ■

Funkcijo `responseHarris`, ki izračuna odziv Harrisovega detektorja oglišč lahko zapišemo kot:

```
1 def responseHarris( iImage, iKappa, iSigma ):
2     """Odziv Harrisovega detektorja oglišč"""
3     iImage = iImage.astype('float')
4     # odvod slike
5     oGx, oGy = imageGradient( iImage )
6     # komponente matrike M
7     A = oGx**2
```

```

8     B = oGy**2
9     C = oGx * oGy
10    # Gaussovo jedro za iSigma
11    iGaussKernel = discreteGaussian2D( iSigma )
12    # glajenje komponent matrike M
13    A = ni.convolve( A, iGaussKernel, mode='nearest' )
14    B = ni.convolve( B, iGaussKernel, mode='nearest' )
15    C = ni.convolve( C, iGaussKernel, mode='nearest' )
16    # odziv Harrisovega detektorja oglišč
17    trM = A + B
18    detM = A * B - C**2
19    oQH = detM - iKappa*(trM**2)
20
21    # PREKO IZRAČUNA LASTNIH VREDNOSTI
22    #     trM = A + B
23    #     detM = A*B - C**2
24    ##     # izračun lastnih vrednosti
25    #     D = (trM/2.0)**2 - detM
26    #     D[D<0] = 0; D = np.sqrt(D)
27    #     l1 = trM/2.0 + D
28    #     l2 = trM/2.0 - D
29    #     oQH = l1*l2 - iKappa*(l1 + l2)**2
30
31    return oQH
32
33 # preizkus funkcije
34 showImage(iImageG)
35 oQH = responseHarris(iImageG, 1/32.0, 3.0)
36 showImage(oQH, 'Odziv Harrisovega detektorja oglišč')

```

Izračun temelji na analizi lastnih vrednosti strukturnih tenzorjev. Najprej izračuna gradient slike v x - in y -smeri, nato pa iz gradientov oblikuje komponente strukturnih tenzorjev (A , B , C). Te komponente se zgladijo z Gaussovim filtrom, kar omogoča robustnejšo detekcijo oglišč pri različnih skalah. Ključni del algoritma je izračun Harrisovega odziva $Q_H = \det(M) - \kappa \cdot \text{tr}(M)^2$, kjer je M strukturni tenzor, $\det(M)$ njegova determinanta, $\text{tr}(M)$ sled tenzorja, κ pa empirični parameter, ki določa občutljivost detektorja. Alternativno (*zakomentirano*) je mogoče odziv izračunati preko lastnih vrednosti strukturnih tenzorjev, kar daje enakovredne rezultate. Visoke vrednosti odziva označujejo lokacije potencialnih oglišč v sliki. Rezultat prikazuje slika 6.7.

Vaja 6.3 Napišite funkcijo za iskanje lokalnih maksimumov v poljubnem 2D polju `iArray`:

```

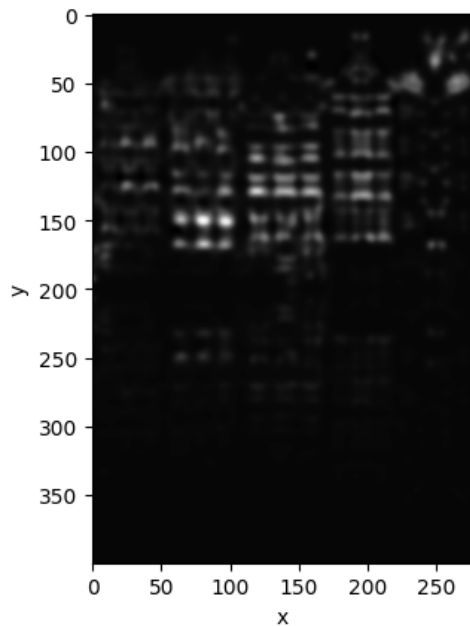
1 def findLocalMax( iArray ):
2     # Tu napišite kodo funkcije
3     return oLocalMax

```

Funkcija vrne matriko `oLocalMax` dimenzij $2 \times n$, ki vsebuje (x,y) koordinate n lokalnih maksimumov. Lokalni maksimumi v 2D polju so tiste točke, v katerih je vrednost večja od vrednosti v sosednjih 8 elementih slike. Pri tem ne upoštevajte elementov slike, ki ležijo na robu ali izven slike. ■

Lokalni maksimumi v odzivu Harrisovega detektorja predstavljajo potencialna oglišča na vhodni sliki. Lokalni maksimumi v 2D polju so tiste točke, v katerih je vrednost večja od vrednosti v sosednjih 8 elementih slike. Rešitev lahko zapišemo:

Odziv Harrisovega detektorja oglišč



Slika 6.7: Prikaz odziva Harrisovega detektorja oglišč $Q_H(x,y)$ za vrednost $\kappa = 1/32$ in Gaussovim glajenjem s $\sigma = 3$.

```

1 def findLocalMax( iArray, iThreshold=None ):
2     """Poišči lokalne optimume"""
3     iArray = np.asarray( iArray )
4     dy, dx = iArray.shape
5     oLocalMax = [] # inicializiraj prazen seznam optimumov
6     for y in range(1,dy-1):
7         for x in range(1,dx-1):
8             cval = iArray[y,x]
9             # preskoči element, če je cval manjše od iThreshold
10            if iThreshold is not None:
11                if cval < iThreshold:
12                    continue
13            # preveri vse elemente v 3x3 soseščini
14            gx, gy = np.meshgrid((x-1,x,x+1), (y-1,y,y+1), sparse=False)
15            gx = gx.flatten()
16            gy = gy.flatten()
17
18            cvaldiff = iArray[gy, gx] - cval
19            cvaldiff[int(np.floor(len(gy)/2))] = -1
20            if cvaldiff.max() < 0:
21                oLocalMax.append( (x,y) )
22        # vrni seznam maksimumov
23    return np.array( oLocalMax )
24
25 # preizkus funkcije
26 oLocalMax = findLocalMax( oQH, 1e+6 )
27 print('Število lokalnih maksimumov: ', str(oLocalMax.shape) )
28 plt.imshow(iImageG, cmap = cm.Greys_r) # prikaži sliko v novem oknu
29 plt.suptitle('Sivinska slika in oglišča') # nastavi naslov slike

```

```

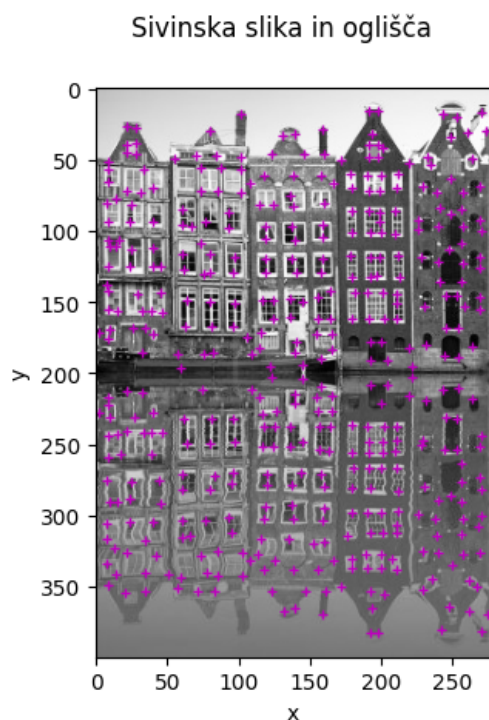
30 plt.xlabel('x')
31 plt.ylabel('y')
32 plt.plot(oLocalMax[:,0],oLocalMax[:,1],'+m',markersize=5.0)
33 plt.show(block=True)

```

Algoritem preverja vsak element v polju (razen robnih) in ga primerja z njegovo 3×3 sosesčino. Če je vrednost elementa večja od vseh sosedov in preseže opcijski prag (`iThreshold`), se lokacija elementa doda v seznam lokalnih maksimumov. Ključna značilnost te implementacije je uporaba maskiranja centralnega elementa v sosesčini, kar omogoča pravilno detekcijo strogo lokalnih maksimumov. Funkcija je še posebej uporabna za detekcijo značilnih točk v odzivnih mapah detektorjev kot so Harrisov detektor ali LoG, kjer lokalni maksimumi predstavljajo potencialne lokacije interesnih točk. Preizkus funkcije za vhodno sivinsko sliko 6.6a vrne:

Število lokalnih maksimumov: (433, 2)

Prikaz lokacije lokalnih maksimumov je na sliki 6.8.



Slika 6.8: Rezultat iskanja lokalnih optimumov v $Q_H(x,y)$ Harrisovega detektorja oglišč za vrednost $\kappa = 1/32$ in Gaussovim glajenjem s $\sigma = 3$.

Vaja 6.4 Napišite funkcijo za detekcijo oglišč na vhodni sivinski sliki `iImage`:

```

1 def cornerHarris( iImage, iKappa, iTmin ):
2     # Tu napišite kodo funkcije
3     return oCorners

```

kjer je `iKappa` občutljivost detektorja oglišč, `iTmin` pa prag za detekcijo oglišč. Funkcija vrne matriko `oCorners` dimenzij $2 \times n$, ki vsebuje (x,y) koordinate n detektiranih oglišč. Pri

tem uporabite funkciji `responseHarris()` in `findLocalMax()`. Parameter `iSigma` v funkciji `responseHarris()` postavite na vrednost 3. ■

Funkcija integrira klica funkcij, ki smo jih napisali pri prejšnjih dveh vajah 6.2 in 6.3 in izpostavi vplivne parametre. Zapišemo jo kot:

```
1 def cornerHarris( iImage, iKappa, iTmin ):
2     """Izloči robne točke s Harrisovim detektorjem oglišč"""
3     iSigma = 3.0
4     oQht = responseHarris( iImage, iKappa, iSigma )
5     oCorners = findLocalMax( oQht, iTmin )
6     return oCorners
7
8 # preizkus funkcije
9 oCorners = cornerHarris(iImageG, 1/32.0, 1e+6)
10 plt.imshow(iImageG, cmap = cm.Greys_r) # prikazi sliko v novem oknu
11 plt.suptitle('Sivinska slika in oglišča') # nastavi naslov slike
12 plt.xlabel('x')
13 plt.ylabel('y')
14 plt.plot(oCorners[:,0],oCorners[:,1],'+m',markersize=5.0)
```

Rezultat je prikazan na sliki 6.8. Dobljen seznam oglišč lahko še nadalje prečistimo, pri čemer so možne strategije omejitve maksimalnega števila oglišč, minimalno evklidska razdalja med oglišči, ipd.

6.6.3 Cannyev detektor robov

Cannyjev detektor boste uporabili za izločanje robov iz slike, na podlagi robov pa boste kasneje s pomočjo Houghovega algoritma poiskali bodisi premice, bodisi kroge, ali pa druge parametrične oblike na sliki robov.

Vaja 6.5 Napišite funkcijo za izračun magnitude `oGAbs` in orientacije `oGPhi` 1. odvoda glajene sivinske slike `iImage`:

```
1 def smoothImageGradientAbs( iImage, iSigma ):
2     # Tu napišite kodo funkcije
3     return oGAbs, oGPhi
```

kjer je `iSigma` standardna deviacija Gaussove funkcije za glajenje slike. Funkcijo preizkusite na sliki `slika3.jpg`, ki ste jo prej pretvorili v sivinsko sliko. Prikažite sliko magnitude 1. odvoda pri vrednosti `iSigma` je 1. ■

Rešitev lahko zapišemo:

```
1 def smoothImageGradientAbs( iImage, iSigma ):
2     iImage = iImage.astype('float')
3     # dušenje šuma z glajenjem
4     if iSigma > 0:
5         iGaussKernel = discreteGaussian2D( iSigma )
6         oImage = ni.convolve( iImage, iGaussKernel, mode='nearest' )
7     else:
8         oImage = iImage
9     # izračun gradienta slike (magnituda + orientacija)
10    oGx, oGy = imageGradient( oImage )
11    oGAbs = np.sqrt(oGx**2.0 + oGy**2.0)
```

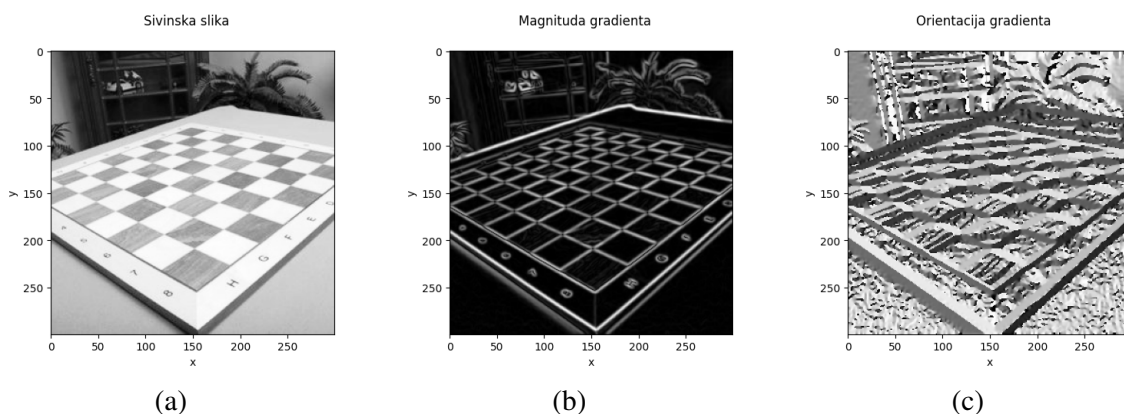


```

12     oGPhi = np.arctan2(oGy, oGx)
13
14     return oGAbs, oGPhi
15
16 # preizkus funkcije
17 iImageG = colorToGray(loadImage('slika3.jpg'))
18 showImage(iImageG)
19 oGAbs, oGPhi = smoothImageGradientAbs(iImageG, 1.0)
20 showImage(oGAbs, 'Magnituda gradienta')
21 showImage(oGPhi, 'Orientacija gradienta')

```

Funkcija `smoothImageGradientAbs` izračuna glajen gradient slike z uporabo Gaussovega filtra in nato določi magnitudo ter orientacijo gradienta. Najprej zgladi vhodno sliko z Gaussovimi filtri podane standardne deviacije (`iSigma`), če je ta vrednost večja od 0. Nato izračuna gradient slike v x in y smeri s funkcijo `imageGradient`. Magnitudo gradienta (`oGAbs`) določi kot evklidsko normo parcialnih odvodov ($\sqrt{G_x^2 + G_y^2}$), orientacijo gradienta (`oGPhi`) pa kot arkus tangens razmerja parcialnih odvodov ($\arctan2(G_y, G_x)$). Rezultat prikazuje slika 6.9.



Slika 6.9: Prikaz (a) sivinske slike in pripadajoči sliki (b) magnitude in (c) orientacije gradienta.

Vaja 6.6 Napišite funkcijo za odstranjevanje nemaksimalnih vrednosti s pomočjo vhodnih slik `oGAbs` in `oGPhi`:

```

1 def nonMaximaSuppression( iGAbs, iGPhi ):
2     # Tu napišite kodo funkcije
3     return oEdge

```

Funkcija naj vrne sliko `oEdge`, kjer naj elementi z vrednostjo 1 predstavljajo nepovezane robove v sliki. Funkcijo preizkusite na slikah 1. odvoda, ki ste jih predhodno izračunali s funkcijo `smoothImageGradientAbs`. ■

Funkcija `nonMaximaSuppression` izvaja postopek odstranjevanja nemaksimalnih vrednosti, ključni korak v Cannyjevem detektorju robov. Najprej normalizira orientacije gradientov na interval $[0, \pi]$ in jih kvantizira v štiri glavne kvadrante orientacije, ki so določeni s smermi (0° , 45° , 90° , 135°). Nato za vsak piksel preveri, ali je njegova magnituda gradienta maksimalna v smeri gradienta v primerjavi z dvema sosednjima pikseloma v tej smeri. Le tisti pikseli, ki izpolnjujejo ta pogoj, so označeni kot potencialni robovi (vrednost 1), medtem ko vsi ostali pikseli dobijo vrednost 0.

Rešitev lahko zapišemo:

```

1 def nonMaximaSuppression( iGAbs, iGPhi ):
2     # pretvori orientacijo gradienta na interval 0-2pi
3     iGPhi =iGPhi.flatten();
4     iGPhi[iGPhi<0] = iGPhi[iGPhi<0] + np.pi;
5
6     # pretvori zvezni kot v indeks enega izmed štirih kvadrantov
7     distPi0 = np.minimum(iGPhi,np.abs(iGPhi-np.pi));
8     distPi45 = np.abs(iGPhi-np.pi/4);
9     distPi90 = np.abs(iGPhi-np.pi/2);
10    distPi135 = np.abs(iGPhi-3*np.pi/4);
11    minIdx = np.argmin(np.vstack((distPi0, distPi45, distPi90, distPi135)),
12                        axis=0)
13    iGPhi = np.reshape(minIdx,iGAbs.shape)
14
15    # poišči maksimalne vrednosti gradienta
16    oEdge = np.zeros_like(iGAbs)
17    dy, dx = oEdge.shape
18    for y in range(1,dy-1):
19        for x in range(1,dx-1):
20            if iGPhi[y,x] == 0:
21                if iGAbs[y,x]>iGAbs[y,x-1] and iGAbs[y,x]>iGAbs[y,x+1]:
22                    oEdge[y,x] = 1;
23            elif iGPhi[y,x] == 1:
24                if iGAbs[y,x]>iGAbs[y-1,x-1] and iGAbs[y,x]>iGAbs[y+1,x+1]:
25                    oEdge[y,x] = 1;
26            elif iGPhi[y,x] == 2:
27                if iGAbs[y,x]>iGAbs[y-1,x] and iGAbs[y,x]>iGAbs[y+1,x]:
28                    oEdge[y,x] = 1;
29            elif iGPhi[y,x] == 3:
30                if iGAbs[y,x]>iGAbs[y-1,x+1] and iGAbs[y,x]>iGAbs[y+1,x-1]:
31                    oEdge[y,x] = 1;
32
33    return oEdge
34
35 # preizkus funkcije
36 oGAbs, oGPhi = smoothImageGradientAbs(iImageG, 1.0)
37 oEdge = nonMaximaSuppression(oGAbs, oGPhi)
38 showImage(oEdge, 'Slika robov po odstranjevanju nemaksimalnih vrednosti')
```

Ta postopek ohranja samo najmočnejše robove v sliki, hkrati pa odstrani šibkejša odzive in šum, kar izboljša kvaliteto detekcije robov. Rezultat prikazuje slika 6.10.

Vaja 6.7 Napišite funkcijo za povezovanje robov vhodni binarni sliki robov iEdge:

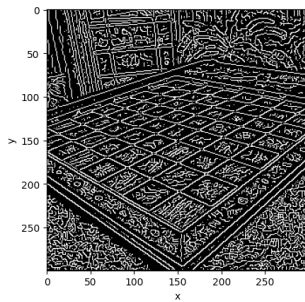
```

1 def connectEdge( iEdge, iGAbs, iThreshold):
2     # Tu napišite kodo funkcije
3     return oEdge
```

kjer je iGAbs slika magnitude gradienta, iThreshold pa dvovrstični vektor z zgornjim in spodnjim pragom za detekcijo izrazitih in neizrazitih robov. Funkcija vrne binarno sliko prečiščenih in povezanih robov oEdge enakih dimenzij kot je vhodna binarna slika iEdge. ■

Funkcija connectEdge implementira naslednji ključni korak Cannyjevega detektorja robov -

Slika robov po odstranjevanju nemaksimalnih vrednosti



Slika 6.10: Prikaz binarne slike robov po odstranitvi točk z nemaksimalno vrednostjo magnitude v smeri gradienta.

povezovanje robov z uporabo histereze. Najprej normalizira magnitudo gradienta na interval $[0, 1]$, saj je tako nastavitev spodnjega in gornjega praga standardizirana. Nato loči robove na izrazite (nad zgornjim pragom) in neizrazite (med spodnjim in zgornjim pragom). Iterativno povezuje neizrazite robove z izrazitimi, če so v njihovi 3×3 soseščini. Postopek se iterativno ponavlja, dokler se množici izrazitih in neizrazitih robov ne stabilizirata. Rešitev zapišemo:

```

1 def connectEdge( iEdge, iGAbs, iThreshold):
2     # upravljanje po magnitudi gradienta s histerezo oz. dvojnimi pragom
3     iGAbs = iGAbs / np.max(iGAbs[iEdge>0]); # normaliziramo med 0 in 1
4     iGAbs[iGAbs>1] = 1;
5     # vrne seznam neizrazitih (weak) in izrazitih (strong) robov
6     edgeWeakIdx = np.nonzero(iEdge*(iGAbs>iThreshold[0]))*(iGAbs<=iThreshold[1]))
7     edgeStrongIdx = np.nonzero(iEdge*(iGAbs>iThreshold[1]))
8
9     # povezovanje neizrazitih z izrazitimi robovi
10    oEdge = np.zeros_like(iEdge)
11    oEdge[edgeStrongIdx] = 1;
12    yxl = np.asarray(edgeWeakIdx)
13    oEdgeP = np.zeros_like(oEdge)
14
15    iterN = 0
16    while np.sum(np.abs(oEdge-oEdgeP)) > 0:
17        iterN = iterN + 1
18        print(iterN)
19
20        oEdgeP = np.copy(oEdge)
21
22        for i in range(yxl.shape[1]):
23            x = yxl[1,i]
24            y = yxl[0,i]
25
26            if oEdge[y,x] > 0:
27                continue
28
29            if x > 0 and y > 0 and y < oEdge.shape[0]-1 and x <
                oEdge.shape[1]-1:
30
31                gx, gy = np.meshgrid((x-1,x,x+1), (y-1,y,y+1), sparse=False)
32                gx = gx.flatten()
33                gy = gy.flatten()
34

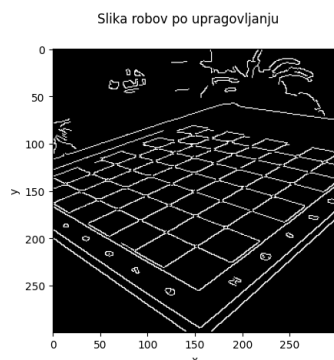
```

```

35         if np.sum(oEdge[gy, gx] == 1) > 0:
36             oEdge[y,x] = 1;
37
38     return oEdge
39
40 # preizkus funkcije
41 oGAbs, oGPhi = smoothImageGradientAbs(iImageG, 1.0)
42 oEdge = nonMaximaSuppression(oGAbs, oGPhi)
43 oEdge = connectEdge( oEdge, oGAbs, [0.15, 0.3])
44 showImage(oEdge, 'Slika robov po upravljanju s histerezo in povezovanju robov')

```

Ta pristop omogoča ohranjanje kontinuitete robov, hkrati pa zatira šum in šibke nepovezane odzive, kar vodi k robustni detekciji pomembnih robov v sliki. Rezultat prikazuje slika 6.11.



Slika 6.11: Prikaz binarne slike robov 6.10 po upravljanju s histerezo in povezovanju robov.

Vaja 6.8 . Napišite funkcijo za detekcijo robov v vhodni sivinski sliki `iImage` z uporabo Cannyjevega algoritma:

```

1 def edgeCanny( iImage, iSigma, iThreshold=0 ):
2     # Tu napišite kodo funkcije
3     return oEdge

```

kjer je `iThreshold` prag, s katerim določimo prave robove slik. Funkcija naj uporabi funkciji `smoothImageGradientAbs` in `nonMaximaSuppression`, ki jima dodamo upravljanje na način, da ohranimo samo tiste detektirane robove z odstranjenimi nemaksimalnimi vrednostmi in magnitudo 1. odvoda `oGAbs` večjo od `iThreshold`. Sliko magnitude 1. odvoda predhodno normalizirajte na interval med 0 in 1 glede na minimalno in maksimalno vrednost. Primerjajte sliko robov za različne vrednosti `iSigma` in `iThreshold`. ■

Funkcija povezuje funkcije, ki smo jih napisali pri prejšnjih vajah 6.5, 6.6 in 6.7. Rešitev zapišemo:

```

1 def edgeCanny( iImage, iThreshold, iSigma ):
2     oGAbs, oGPhi = smoothImageGradientAbs(iImage,iSigma)
3     oEdge = nonMaximaSuppression(oGAbs, oGPhi)
4     oEdge = connectEdge( oEdge, oGAbs, iThreshold)
5
6     return oEdge
7

```

```

8 | # preizkus funkcije
9 | iThreshold = [0.15, 0.3]
10 | oEdge = edgeCanny(iImageG, iThreshold, 1)
11 | showImage(oEdge, 'Slika robov po upragovljanju')

```

Rezultat prikazuje slika 6.11.

6.6.4 Zaznavanje premic s Houghovo preslikavo

Vaja 6.9 S pomočjo Houghove preslikave za parametrični model premice, ki je že implementirana v knjižnici OpenCV, poiščite vse premice, ki jih določajo robovi na sliki *slika3.jpg* (podobno kot slika 6.11). Implementacija Houghove preslikave sama izvede izločanje robov iz dane slike, le ustrezne parametre je potrebno podati:

```

1 | lines = HoughLines( iImage, rho, theta, threshold )

```

kjer polje `lines` vsebuje pare parametrov r in ϕ , ki določata položaj premice na sliki. Prikažite vse premice na sliki. ■

Rešitev zastavimo tako, da najprej naložimo vhodno sliko, zaznamo robove, nato pa z funkcijo `cv.HoughLines()` zaznamo premice, ki potekajo skozi *ravne robove* v sliki. Parametri v funkciji `cv.HoughLines()` določajo ločljivost akumulatorja v polarne koordinate (0.5 pikslov za ρ in 0.5° za θ) ter prag za število glasov (60; robovi vsaj dolžine 60 pikslov). Za vsako zaznano črto v polarne koordinate (ρ , θ) koda izračuna dve točki na črti v kartezijskem koordinatnem sistemu, ki sta oddaljeni 1000 pikslov od osnovne točke (x_0, y_0). Koordinate teh točk pretvori v celoštevilске vrednosti in na sliko nariše rdečo črto debeline 1 piksel. Na koncu pretvori barvni prostor iz BGR v RGB za pravi prikaz.

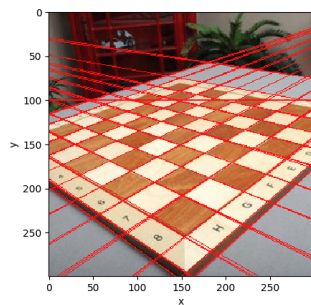
```

1 | img = cv.imread('slika3.jpg')
2 | edges = cv.Canny(imgG, 75, 151) # implementacija v OpenCV
3 | lines = cv.HoughLines(edges, 0.5, 0.5*np.pi/180, 60)
4 | for line in lines:
5 |     for rho, theta in line:
6 |         a = np.cos(theta)
7 |         b = np.sin(theta)
8 |         x0 = a*rho
9 |         y0 = b*rho
10 |        x1 = int(x0 + 1000*(-b))
11 |        y1 = int(y0 + 1000*a)
12 |        x2 = int(x0 - 1000*(-b))
13 |        y2 = int(y0 - 1000*a)
14 |
15 |        cv.line(img, (x1, y1), (x2, y2), (0, 0, 255), 1)
16 |
17 | rgb = cv.cvtColor(iImage, cv.COLOR_BGR2RGB)
18 |
19 | showImage(rgb)

```

Rezultat prikazuje slika 6.12.

6.6.5 Konvolucijske nevronske mreže



Slika 6.12: Prikaz premic, ki jih zaznamo s pomočjo Houghove preslikave na podlagi binarne slike robov 6.10.

Vaja 6.10 Konvolucijske nevronske mreže se uporabljajo tudi za razpoznavo objektov v sliki. Model GoogLeNet je konvolucijska nevronska mreža (CNN), ki izvaja klasifikacijo slik na podlagi nabora podatkov ImageNet z 1000 različnimi razredi. Delovanje poteka v treh glavnih fazah:

1. **Predobdelava slike:** Vhodna slika se preoblikuje v blob format z dimenzijami 224x224 pikslov in normalizira z odštevanjem srednjih vrednosti (104, 117, 123) za vsak barvni kanal. Ta korak zagotavlja, da so vhodni podatki primerljivi s podatki, na katerih je bil model učen.
2. **Napovedovanje:** Model vrne vektor verjetnosti (preds) dolžine 1000, kjer vsak element predstavlja verjetnost, da vhodna slika pripada določenemu razredu iz ImageNet nabora.
3. **Interpretacija rezultatov:** Izhodne vrednosti so:
 - preds[0]: Vektor verjetnosti za vse razrede (1000 vrednosti)
 - idxs: Indeksi najverjetnejših razredov, urejeni padajoče
 - classes[idx]: Človeku berljiva oznaka razreda
 - preds[0][idx] * 100: Verjetnost v procentih

Že naučeno nevronske mreže GoogLeNet v knjižnici OpenCV uporabite na danih oz. svojih slikah in preverite njeno natančnost razpoznavanja objektov. ■

Rešitev lahko zapišemo:

```
1 # uvoz potrebnih knjižnic
2 import numpy as np
3 import time
4 import cv2
5
6 # nastavitve poti do vhodnih datotek
7 args = {"image": "slika6.png",
8         "prototxt": "bvlc_googlenet.prototxt",
9         "model": "bvlc_googlenet.caffemodel",
10        "labels": "synset_words.txt"}
11
12 # naloži vhodno sliko z diska
13 image = cv2.imread(args["image"])
14
15 # naloži oznake razredov iz datoteke
16 rows = open(args["labels"]).read().strip().split("\n")
17 classes = [r[r.find(" ") + 1:].split(",")[0] for r in rows]
18
19 # nevronska mreža zahteva fiksne dimenzije vhodne slike (224x224 pikslov)
```



```

20 # in normalizacijo z odštevanjem povprečnih vrednosti (104, 117, 123)
21 # po izvedbi tega ukaza ima "blob" obliko (1, 3, 224, 224)
22 blob = cv2.dnn.blobFromImage(image, 1, (224, 224), (104, 117, 123))
23
24 # naloži prednaučen model z diska
25 print("[INFO] nalagam model...")
26 net = cv2.dnn.readNetFromCaffe(args["prototxt"], args["model"])
27
28 # nastavi blob kot vhod mreže in izvedi klasifikacijo
29 net.setInput(blob)
30 start = time.time()
31 preds = net.forward()
32 end = time.time()
33 print("[INFO] klasifikacija je trajala {:.5} sekund".format(end - start))
34
35 # uredi indekse verjetnosti v padajočem vrstnem redu in izberi top-5 napovedi
36 idxs = np.argsort(preds[0])[::-1][:5]
37
38 # prikaži top-5 napovedi
39 for (i, idx) in enumerate(idxs):
40     # nariši najboljšo napoved na sliko
41     if i == 0:
42         text = "Oznaka: {}, {:.2f}%".format(classes[idx],
43         preds[0][idx] * 100)
44         cv2.putText(image, text, (5, 25), cv2.FONT_HERSHEY_SIMPLEX,
45         0.7, (0, 0, 255), 2)
46
47     # izpiši napovedano oznako in verjetnost
48     print("[INFO] {}. oznaka: {}, verjetnost: {:.5}".format(i + 1,
49     classes[idx], preds[0][idx]))
50
51 # prikaži rezultate na sliki
52 cv2.imshow("Slika", image)
53 cv2.waitKey(0)

```

Model vrne naslednje izhodne vrednosti:

- **Verjetnost (preds[0][idx]):** kaže stopnjo zaupanja modela, da slika pripada določenemu razredu. Vrednosti so med 0 (nič verjetno) in 1 (popolnoma verjetno). Običajno upoštevamo samo vrednosti nad 0.5 (50%).
- **Oznaka razreda (classes[idx]):** besedilni opis razreda (npr. "orel", "avtomobil"). Te oznake se naložijo iz datoteke `synset_words.txt` in so vnaprej določene z naučenim modelom.
- **Čas klasifikacije:** čas v sekundah, ki ga model potrebuje za obdelavo slike, je ključen indikator zmogljivosti za realnočasovne aplikacije.

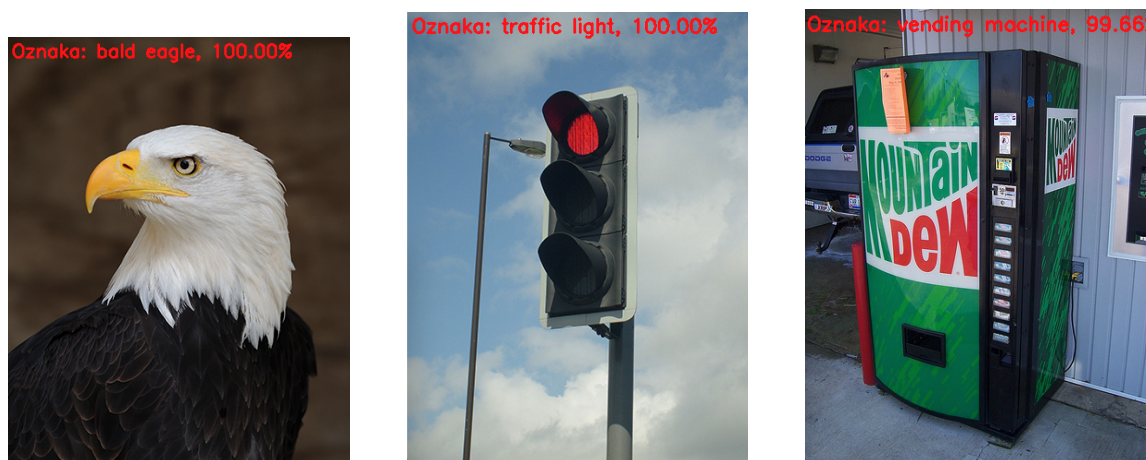
Model za dano vhodno sliko na izhodu razvrsti oznake glede na padajočo verjetnost in izpiše pet najbolj verjetnih oznak ter pripadajoče verjetnosti:

```

[INFO] nalagam model...
[INFO] klasifikacija je trajala 0.039151 sekund
[INFO] 1. oznaka: bald eagle, verjetnost: 0.99997
[INFO] 2. oznaka: kite, verjetnost: 3.4535e-05
[INFO] 3. oznaka: vulture, verjetnost: 1.6599e-07
[INFO] 4. oznaka: hornbill, verjetnost: 2.8287e-08
[INFO] 5. oznaka: albatross, verjetnost: 2.4403e-08

```

Slika se prikaže v novem oknu in oznaka z najvišjo verjetnostjo se izpiše kot prikazujejo primeri na sliki 6.13.



Slika 6.13: Prikaz zaznave objektov z GoogLeNet z izpisom oznake in pripadajoče verjetnosti.

6.7 Naloge in vprašanja

Poročilo naloge pripravite v obliki Python skripte, ki naj izvede zahtevane izračune oz. klice funkcij in izriše pripadajoče slike oz. grafe. Vaša dolžnost je, da v skripti uvozite ustrezne Python knjižnice in dodate morebitne podporne skripte. Skripta se mora izvesti in poustvariti rezultate, kot se zahtevajo v nalogah. Na vprašanja odgovorite v obliki komentarja, naprimer *# Naloga X.2: To je moj odgovor..*

6.7.1 Gradient sivinske slike

Naloga 6.1 Izračunajte druge odvode sivinske slike $f(x,y)$: $\partial^2 f(x,y)/\partial x^2$, $\partial^2 f(x,y)/\partial y^2$, $\partial^2 f(x,y)/\partial x\partial y$ in $\partial^2 f(x,y)/\partial y\partial x$. Prikažite te odvode kot sivinske slike.

6.7.2 Harrisov detektor oglišč

Naloga 6.2 Razširite funkcijo `cornerHarris()` iz vaje 6.4 tako, da bo imela dodaten vhodni parameter `iMinDist`. Parameter `iMinDist` naj določa minimalno razdaljo med oglišči v slikovnih elementih. Prečistite seznam oglišč tako, da bodo oglišča med seboj oddaljena vsaj za razdaljo `iMinDist`, pri čemer ohranite oglišča z večjim odzivom Harris detektorja. Izrišite sivinsko sliko z vrisanimi detektiranimi oglišči, in sicer za vrednost `iMinDist = 0` in `iMinDist = 20`.

Naloga 6.3 Uporabite OpenCV funkcijo `cornerHarris` za izračun odziva Harrisovega detektorja oglišč na sliki `slika1.jpg`:

```
1 | oQH = cornerHarris( iImage, blockSize, ksize, k )
```

kjer je k občutljivost detektorja oglišč (κ), `blockSize` pa standardna deviacija Gaussove funkcije za glajenje odziva. Funkcija vrne sliko `oQH`, ki predstavlja odziv detektorja oglišč. Preizkusite delovanje funkcije za različne vrednosti parametrov `k` in `blockSize`. Uporabite upravljanje za določitev območij visokih odzivov. Primerjajte dobljene rezultate s funkcijo, ki ste jo implementirali pri vaji 6.4.

Naloga 6.4 Detekcijo oglišč običajno kombiniramo z različnimi strategijami prečiščevanja seznama oglišč, kar implementira naslednja funkcija v knjižnici OpenCV:

```
1 pts = goodFeaturesToTrack( iImage, maxCorners, qualityLevel,
2                             minDistance, None, None, blockSize,
3                             useHarrisDetector, k )
```

kjer vrnjeno Polje pts vsebuje koordinate detektiranih oslonilnih točk oz. oglišč. Funkcija `goodFeaturesToTrack()` iz knjižnice OpenCV implementira algoritem za detekcijo robustnih oglišč, ki temelji na Shi-Tomasijevem ali Harrisovem detektorju oslonilnih točk oziroma oglišč, v vhodni sivinski sliki. Izbere oglišča z največjo lastno vrednostjo v izračunu strukturnih tenzorjev, kar zagotavlja robustno detekcijo oglišč, ki so optimalni za aplikacijo sledenja, ipd. Ključni parametri funkcije vključujejo maksimalno število točk, minimalno kakovost in minimalno evklidsko razdaljo med točkami. Algoritem je še posebej uporaben za inicializacijo algoritmov kot je Lukas-Kanade algoritem sledenja (glej poglavje 11) ali za izbor značilnih točk pri strukturi iz gibanja (glej poglavje 12), saj zagotavlja dobro prostorsko porazdelitev stabilnih oslonilnih točk (oz. oglišč) po celotni vhodni sliki.

Preučite dokumentacijo funkcije in ustrezno nastavite parametre tako, da boste dobili seznam oglišč kot pri vaji 6.4. Primerjajte rezultat tudi z rezultatom in naloge 6.3.

Naloga 6.5 Z drugačno kriterijsko funkcijo, kot jo za analizo lastnih vrednosti λ_1 in λ_2 matrike M , $\lambda_1 > \lambda_2$, uporablja Harrisov detektor oglišč, lahko iščemo, detektiramo ali poudarjamo podolgovate strukture v sliki. Ena od tovrstnih funkcij je *linearna anizotropičnost*:

$$Q_{LA} = \frac{\lambda_1 - \lambda_2}{\lambda_1 + \lambda_2 + \beta}, \quad (6.10)$$

kjer je $\beta > 0$ poljubna konstanta. Napišite funkcijo za poudarjanje podolgovatih struktur v vhodni sivinski sliki `iImage`:

```
1 def enhanceLinear( iImage, iSigma, iBeta ):
2     # Tu napišite python kodo funkcije
3     return oQLA
```

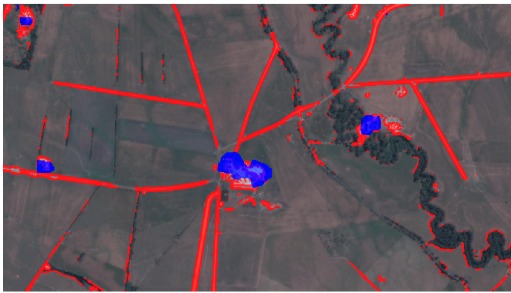
kjer je `iSigma` standardna deviacija Gaussove funkcije za glajenje elementov matrike M , `iBeta` pa poljubna nenegativna konstanta. Nesortirane lastne vrednosti Harrisovega detektorja lahko dobite z OpenCV funkcijo `cornerEigenValsAndVecs`.

Naložite barvno *RGB* sliko `slika5.png`, jo pretvorite v sivinsko sliko in nelinearno skalirajte z vrednostjo $\gamma = 2$. Uporabite funkcijo `enhanceLinear()` za poudarjanje cest na satelitski sliki tako, da določite optimalne vrednosti parametrov `iSigma` in `iBeta`. Prikažite po vašem optimalen odziv `oQLA` s poudarjenimi podolgovatimi strukturami in izpišite izbrani vrednosti `iSigma` in `iBeta`. Z upravljanjem odziva izluščite masko cest in jih na barvni sliki pobarvajte z rdečo barvo.

S primerno izbiro parametrov lahko Harrisov detektor uporabimo tudi za zaznavanje okroglih struktur. Na isti sivinski in nelinearno skalirani sliki s funkcijo `cornerHarris` izluščite območja objektov in območja brez rastja. Pridobljeni odziv prikažite, njegovo razmejitev pa uporabite na barvni sliki za obarvanje teh območij z modro barvo. Ena izmed možnih rešitev je prikazana na sliki 6.14.

6.7.3 Cannyev detektor robov

Naloga 6.6 Zgornji in spodnji prag za povezovanje robov pri Canny-jevem detektorju robov lahko avtomatsko izračunamo na podlagi statistične analize sivin v sliki. Napišite funkcijo za detekcijo robov:



Slika 6.14: Prikaz rezultata filtriranja za poudarjanja podolgovatih struktur v slikah, ki temelji na linearni anizotropičnosti.

```
1 def canny( iImage, thr_type ):
2     # Tu napišite python kodo funkcije
3     return oEdge
```

ki avtomatsko izračuna spodnji in zgornji prag za povezovanje robov in ju nato uporabi za detekcijo s Cannyjevim detektorjem. Z uporabo vhodne spremenljivke `thr_type` ločite med dvema načinoma določanja pragov:

$$\begin{aligned} T_L &= \max(0, (1 - \sigma)m), & T_L &= T_O/2, \\ T_H &= \min(255, (1 + \sigma)m), & T_H &= T_O, \end{aligned} \quad (6.11) \quad (6.12)$$

kjer je m mediana sivin v sliki, $\sigma = 1/3$ in T_O je prag pridobljen z Otsu upravljanjem. Otsu upravljanje avtomatsko določi prag, ki najbolje razmeji sliko na dva dela. Funkcijo poiščite v knjižnici `OpenCV`. Na treh lastno poiskanih slikah, ki imajo različne zahtevnosti določanja robov, primerjajte oba avtomatska načina izbire z ročno izbranimi pragovoma, ki najbolje izluščita rob. Komentarajte rezultate in uspešnost avtomatskih postopkov. Slike poimenujte kot `canny_###_#.png`, kjer `###` zamenjajte z zadnjimi tremi števkami vaše vpisne številke in `#` z zaporednim številom med 1 in 3.

Naloga 6.7 Uporabite funkcijo v knjižnici `OpenCV` za detekcijo robov v vhodni sivinski sliki `iImage` z uporabo Cannyjevega algoritma:

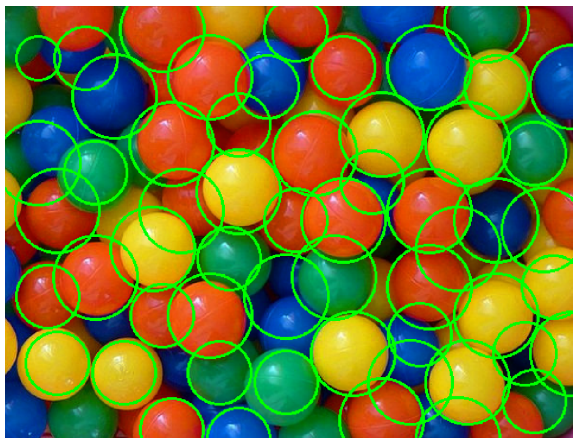
```
1 edges = Canny( iImage, threshold1, threshold2 )
```

kjer praga `threshold1` in `threshold2` določata območja povezovanja robov. Preizkusite različne vrednosti pragov na slikah `slika1.jpg`, `slika2.jpg` in `slika3.jpg`. Primerjajte delovanje funkcije v primerjavi z implementacijo iz vaje 6.8.

6.7.4 Zaznavanje krogov s Houghovo preslikavo

Naloga 6.8 Na vaji 6.9 smo si pogledali Houghovo transformacijo za detekcijo premic v sliki. Na podoben način lahko detektiramo tudi okrogle strukture na sliki. Naložite sliko `slika4.jpg` barvnih kroglic, jo gladite z Gaussovim filtrom iz knjižnice `OpenCV` z velikostjo jedra 3×3 , ter nato uporabite `OpenCV` funkcijo `HoughCircles` za detekcijo središč in radijev krožnic. Kot rezultat detekcije na originalno sliko superponirajte narisane krožnice (funkcija `circle`) detektiranih kroglic. Število detektiranih kroglic tudi izpišite. Na podlagi barvnega tona (iz `HSV` prostora) središča krožnice avtomatsko preštejte število kroglic vsake barve. *Opozorilo:* funkcija `HoughCircles` že sama uporabi Cannyjev detektor robov, praga pa podate kot parametra `param1` in `param2`, zato je

vhodna slika v funkcijo sivinska slika in ne slika robov kot je bilo to na vajah. Pravilno izbiro pragov pa vseeno poiščite s klicem funkcije Canny in prikazom njenega rezultata. Primerjajte rezultat s sliko 6.15.



Slika 6.15: Prikaz rezultata funkcije HoughCircles nad sliko barvnih kroglic slika4.jpg.

6.7.5 Konvolucijske nevronske mreže

Naloga 6.9 Dodatno testirajte uspešnost zaznavanja konvolucijskih nevronskih mrež. Poiščite 5 slik na katerih je postopek zaznave uspešen in 5 slik kjer je zaznava objekta neuspešna. Analizirajte in primerjajte verjetnosti, ki napovedujejo zanesljivost napovedi, med obema množicama slik. Poročajte o ugotovitvah. Slike poimenujte kot `cnn_object_detection_group_###.png`, kjer `group` zamenjajte glede na množico `failed` ali `success`, `###` zamenjajte z zadnjimi tremi številkami vaše vpisne številke ter `#` z ustreznim zaporednim številom med 1 in 5.

Naloga 6.10 Skripto iz datoteke `deep_learning_with_opencv.py` pretvorite v funkcijo:

```
1 def detect_object_with_cnn( params ):
2     # Tu napišite python kodo funkcije
3     return predict_labels, predict_vals
```

Vhodni parameter `params` naj bo dictionary parametrov za detekcijo (trenutna spremenljivka `args`). Funkcija naj vrne seznama petih najverjetnejših opisov objekta slike in pripadajoče verjetnosti (po želji lahko oboje vrnete tudi v kakšni drugi obliki, npr. v eni spremenljivki tipa `dict`).

Napisano funkcijo uvozite v glavno skripto, kjer boste izvajali domače naloge. Tam z uporabo `for` zanke iterirajte skozi seznam poti do vaših 10-ih slik, ustrezno spremenite vhodni dictionary in pokličete funkcijo. V trenutni skripti odstranite zadnji dve vrstici za prikaz slike in čakanje (`waitKey`), saj sta za namene naloge nepotrebna.

7. Prostorske preslikave in poravnave

7.1 Geometrijske preslikave

S prostorskimi ali geometrijskimi preslikavami $\mathcal{T} : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ oz. $\mathcal{T} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ preslikamo slikovne elemente 2D slik (x, y) oz. 3D slik (x, y, z) na nove lokacije (u, v) oz. (u, v, w) , njihove sivinske vrednosti pa pri tem ohranimo. Na ta način lahko izvedemo povečavo oz. pomanjšavo (skaliranje), premik (translacijo), zasuk (rotacijo), pa tudi številne druge linearne in nelinearne geometrijske preslikave slik. Poljubno geometrijsko preslikavo za 2D oz. 3D sliko zapišemo kot:

$$(u, v) = \mathcal{T}(x, y) \quad \text{oz.} \quad (u, v, w) = \mathcal{T}(x, y, z).$$

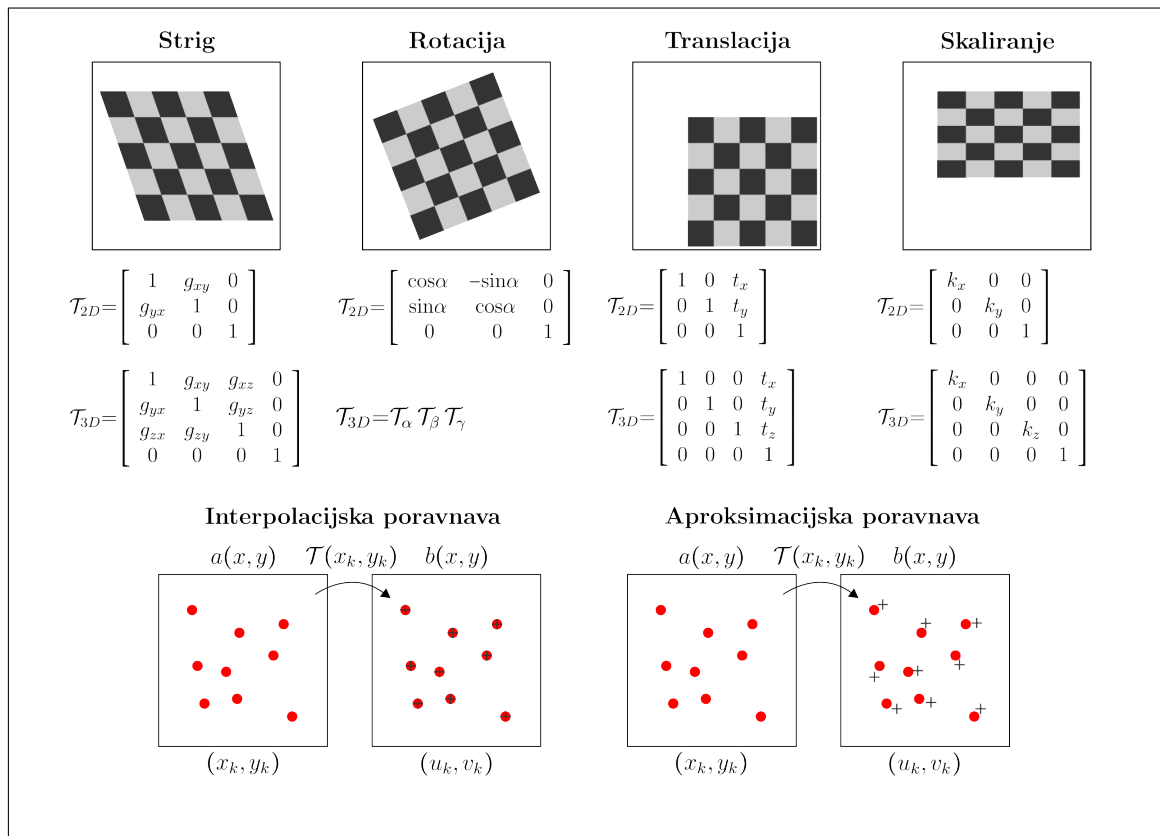
Najbolj splošna linearna geometrijska preslikava je **afina preslikava**, ki omogoča poljubno skaliranje, translacijo, rotacijo in strig. Aфина preslikava je v 2D določena s 6, v 3D pa z 12 parametri:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix}}_{\mathcal{T}_{afina}} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{oz.} \quad \begin{bmatrix} u \\ v \\ w \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} a_{11} & a_{12} & a_{13} & t_x \\ a_{21} & a_{22} & a_{23} & t_y \\ a_{31} & a_{32} & a_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathcal{T}_{afina}} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

kjer parametri t_x, t_y in t_z določajo translacijo v x, y in z smeri slike, parametri a_{ij} pa skaliranje, rotacijo in strig. Matriko za afino preslikavo lahko sestavimo z zaporednim množenjem homogenih matrik posameznih elementarnih preslikav v zelenem vrstnem redu, naprimer:

$$\mathcal{T}_{afina} = \mathcal{T}_{trans} \mathcal{T}_{strig} \mathcal{T}_{rot} \mathcal{T}_{skal}.$$

V praksi se pogosto uporabljata **toga preslikava**, ki jo dobimo z združevanjem rotacijske in translacijske elementarne preslikave in **podobnostna preslikava**, ki jo dobimo z združevanjem toge preslikave in skaliranja. Elementarne preslikave in njihova izražava s homogenimi transformacijskimi matrikami so prikazane v sliki 7.1 zgoraj.



Slika 7.1: Elementarne preslikave in njihova izražava s homogenimi transformacijskimi matrikami (zgoraj) in vpliv interpolacijske in aproksimacijske poravnave (spodaj).

7.2 Poravnava pripadajočih parov točk

Geometrijska poravnava dveh oblik je proces iskanja optimalne geometrijske preslikave $\mathcal{T}$, ki obliki preslika tako, da se iste strukture oz. značilnice oblike nahajajo v enakem ali čim bolj »podobnem« položaju. Značilnice 2D oz. 3D oblik so pogosto kar oblaki točk, zato se za mero podobnosti med dvema oblikama pogosto uporablja povprečna kvadratna Evklidska razdalja med pripadajočimi oz. korespondenčnimi točkami. Na osnovi množice K parov korespondenčnih točk lahko določimo preslikavo $\mathcal{T}$ tako, da se preslikane točke referenčne oblike $\mathcal{T}(x_i, y_i)$ čim bolj prilegajo točkam vhodne oblike (u_i, v_i) in obratno:

$$(u_i, v_i) \leftrightarrow \mathcal{T}(x_i, y_i) \quad \text{oz.} \quad (x_i, y_i) \leftrightarrow \mathcal{T}^{-1}(u_i, v_i).$$

Glede na število parov korespondenčnih točk in vrsto preslikave lahko točke lahko preslikamo tako, da se popolnoma prekrivajo, tj. $\mathcal{T}(x_i, y_i) = (u_i, v_i)$, kar imenujemo **interpolacijska poravnava**, ali pa tako, da se točke le približno prekrijejo, tj. $\mathcal{T}(x_i, y_i) \approx (u_i, v_i)$, kar imenujemo **aproksimacijska poravnava**. Prikaz interpolacijske in aproksimacijske poravnave na poravnavo pripadajočih parov točk je prikazan na sliki 7.1 spodaj.

Afino interpolacijsko poravnavo med 2D oblikami lahko enolično določimo, če poznamo natanko tri pare ($K = 3$) nekolinearnih točk $(u_i, v_i) \leftrightarrow (x_i, y_i)$, poravnavo med 3D oblikami pa če poznamo natanko šest parov ($K = 6$) nekolinearnih točk. V 2D afino interpolacijsko poravnavo $\mathcal{T}$ med tremi pari točk zapišemo kot:

$$\mathcal{T} = \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix}^{-1}.$$

Afino aproksimacijsko poravnavo lahko med 2D oblikami izvedemo, kadar poznamo več kot tri pare ($K > 3$), med 3D oblikami pa več kot šest parov ($K > 6$) korespondenčnih točk. V tem primeru dobimo predoločen sistem enačb, zato korespondenčne točke lahko poravnamo le približno (aproksimacijsko). To storimo tako, da minimiziramo povprečno kvadratno Evklidsko razdaljo med korespondenčnimi točkami:

$$\Sigma^2 = \frac{1}{K} \sum_{i=1}^K \|(u_i, v_i) - \mathcal{T}(x_i, y_i)\|^2.$$

Optimalne vrednosti neznanih parametrov preslikave dobimo tako, da odvode povprečne kvadratne Evklidske razdalje Σ^2 po vseh parametrih postavimo na nič in dobimo sistem enačb za reševanje. Za 2D afino aproksimacijsko poravnavo dobimo naslednji sistem enačb:

$$\underbrace{\begin{bmatrix} \overline{xx} & \overline{xy} & \bar{x} & 0 & 0 & 0 \\ \overline{xy} & \overline{yy} & \bar{y} & 0 & 0 & 0 \\ \bar{x} & \bar{y} & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \overline{xx} & \overline{xy} & \bar{x} \\ 0 & 0 & 0 & \overline{xy} & \overline{yy} & \bar{y} \\ 0 & 0 & 0 & \bar{x} & \bar{y} & 1 \end{bmatrix}}_{\mathcal{P}_{xy}} \underbrace{\begin{bmatrix} a_{11} \\ a_{12} \\ t_x \\ a_{21} \\ a_{22} \\ t_y \end{bmatrix}}_{\mathbf{t}} = \underbrace{\begin{bmatrix} \overline{ux} \\ \overline{uy} \\ \bar{u} \\ \overline{vx} \\ \overline{vy} \\ \bar{v} \end{bmatrix}}_{\mathbf{p}_{uv}},$$

kjer elementi s črto označujejo povprečne vrednosti $\overline{vx} = \frac{1}{K} \sum_{i=1}^K u_i x_i$ oz. $\bar{x} = \frac{1}{K} \sum_{i=1}^K x_i$. Zgornji sistem lahko zapišemo v matrični obliki kot $\mathcal{P}_{xy} \mathbf{t} = \mathbf{p}_{uv}$, kjer vektor $\mathbf{t}$ predstavlja vektorski zapis parametrov affine preslikave $\mathcal{T}$. Parametre affine preslikave dobimo z reševanjem sistema $\mathbf{t} = \mathcal{P}_{xy}^{-1} \mathbf{p}_{uv}$.

Pri poravnavi oblik korespondenčne točke običajno niso vnaprej znane in jih je potrebno določiti v procesu poravnave oblik. Ena od uveljavljenih metod za določanje korespondenčnih točk je **metoda iterativno najbližje točke**¹, pri kateri v vsakem koraku k določimo korespondenčne točke z iskanjem najbližjih parov točk $(u_i, v_i) \leftrightarrow (x_i, y_i)$ in nato izračunamo interpolacijsko ali aproksimacijsko poravnavo $\mathcal{T}_k$ med temi pari točk. Korespondenčne pare točk določimo tako, da vsaka točka (u_i, v_i) pripada kvečjemu enemu paru (x_i, y_i) . Premično obliko preslikamo s $\mathcal{T}_k$ in ponovimo korak. Postopek ponavljamo dokler je relativna sprememba $\|\mathcal{T}_k - I\|_\infty > \varepsilon$ oz. dokler ne izvedemo maksimalno število korakov k_{max} . Če so preslikave $\mathcal{T}_k$ linearne, potem lahko preslikavo med oblikama po k -tem koraku določimo z zaporednim matričnim množenjem vmesnih preslikav:

$$\mathcal{T} = \mathcal{T}_k \mathcal{T}_{k-1} \mathcal{T}_{k-2} \cdots \mathcal{T}_1.$$

7.3 Vaje z rešitvami

Za izvedbo vaje boste uporabili zbirko 2D oblik v datoteki `letala.npy`, ki vsebuje seznam tipa `list` s 30 različnimi oblikami in ki jih v okolje Python naložite z ukazom `numpy.load()`. Oblike so podane z (x, y) koordinatami v `numpy` polju. Pri laboratorijski vaji boste napisali funkcije za 2D afino preslikavo, interpolacijsko in aproksimacijsko poravnavo ter funkcijo za poravnavo oblik z metodo iterativno najbližje točke. Koda vaje je dostopna na [Git repozitoriju](#).

Za izvedbo vaje bomo potrebovali naslednje knjižnice, ki jih ustrezno uvozimo:

```
1 import scipy.ndimage as ni
2 import matplotlib.pyplot as plt
3 import matplotlib.cm as cm
4 import numpy as np
```

¹ang. iterative closest point (ICP)

```
5 | import scipy as sp
```

Vaja 7.1 Napišite funkcijo, ki ustvari 3×3 matriko poljubne 2D afine preslikave:

```
1 | def transAffine2D( iScale, iTrans, iRot, iShear ):
2 |     # Tu napišite python kodo funkcije
3 |     return oMat2D
```

kjer je *iScale* dvovrstični vektor s parametroma skale k_x, k_y , *iTrans* dvovrstični vektor s parametroma translacije t_x, t_y , *iRot* kot rotacije α v stopinjah in *iShear* dvovrstični vektor s parametroma striga g_x, g_y . Funkcija vrne homogeno matriko *oMat2D*, ki ima dimenzije 3×3 . Preverite delovanje funkcije tako, da točke izbrane oblike preslikate s poljubnimi parametri striga, rotacije, translacije in skaliranja, nato pa še s poljubno 2D afino preslikavo. Pri tem poskrbite, da bodo točke zapisane v homogenih koordinatah. Vhodne in preslikane točke oblike prikažite z ukazom `matplotlib.pyplot.plot()`. ■

Funkcijo `transAffine2D( iScale, iTrans, iRot, iShear )`, ki ustvari poljubno 2D afino preslikavo na osnovi kompozituma štirih elementarnih preslikav (striga, rotacije, translacije in skaliranja) lahko zapišemo:

```
1 | def transAffine2D( iScale=(1, 1), iTrans=(0, 0), iRot=0, iShear=(0, 0) ):
2 |     """Funkcija za poljubno 2D afino preslikavo"""
3 |     iRot = iRot * np.pi / 180
4 |     oMatScale = np.matrix( ((iScale[0],0,0),(0,iScale[1],0),(0,0,1)) )
5 |     oMatTrans = np.matrix( ((1,0,iTrans[0]),(0,1,iTrans[1]),(0,0,1)) )
6 |     oMatRot = np.matrix( ((np.cos(iRot),-np.sin(iRot),0),\
7 |                          (np.sin(iRot),np.cos(iRot),0),(0,0,1)) )
8 |     oMatShear = np.matrix( ((1,iShear[0],0),(iShear[1],1,0),(0,0,1)) )
9 |     # ustvari izhodno matriko
10 |    oMat2D = oMatTrans * oMatShear * oMatRot * oMatScale
11 |    return oMat2D
```

Funkcija najprej pretvori kot rotacije iz stopinj v radiane, nato ustvari posamezne transformacijske matrike v homogenih koordinatah. Vsaka preslikava je predstavljena z lastno matriko: *oMatScale* za skaliranje, *oMatTrans* za translacijo, *oMatRot* za rotacijo in *oMatShear* za strig. Funkcija nato te matrike združi z množenjem v vrstnem redu, ki najprej uporabi skaliranje, nato rotacijo, sledi strig in na koncu translacija. Končna homogena 3×3 matrika *oMat2D* združuje vse te preslikave v eno samo preslikavo, ki jo lahko uporabimo za transformacijo 2D točk ali slik. Ključna prednost te implementacije v primerjavi s splošno afino matriko je modularnost - vsako elementarno transformacijo lahko nastavimo neodvisno od drugih. Še bolj splošna implementacija bi lahko vključevala definicijo vrstnega reda elementarnih transformacij. Preizkus funkcije opravimo z nalaganjem in prikazovanjem originalne in preslikane oblike na naslednji način:

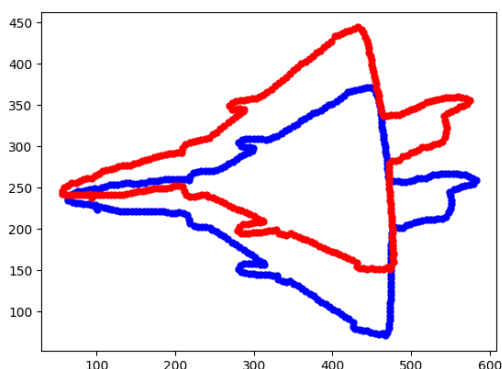
```
1 | # pripravimo pomožno funkcijo za dodajanje homogene koordinate
2 | def addHomo( iPts ):
3 |     iPts = np.hstack( (iPts, np.ones((iPts.shape[0],1))) )
4 |     return iPts
5 |
6 | # naložimo testno obliko (v novejših verzijah je potrebno nastaviti parameter
   |   allow_pickle=True)
7 | letala = np.load('letala.npy', allow_pickle=True)
```

```

8 let = np.matrix(letala[0])
9 # dodamo homogeno koordinato
10 let = addHomo( let )
11 # prikažemo originalno obliko
12 plt.plot(let[:,0],let[:,1], 'ob', markersize=5.0)
13 # preslikamo obliko in jo prikažemo
14 oMat2D = transAffine2D( iScale=(1, 1), iTrans=(10, 0), iRot=10, iShear=(0.1, 0)
15 )
16 let_t = (oMat2D * let.transpose()).transpose()
17 plt.plot(let_t[:,0],let_t[:,1], 'or', markersize=5.0)

```

Rezultat prikazuje slika 7.2.



Slika 7.2: Prikaz originalne oblike (*modra*) in s poljubno afino preslikavo preslikane različice (*rdeča*).

Vaja 7.2 Napišite funkcijo, ki med pari korespondenčnih točk določi 2D afino interpolacijsko poravnavo:

```

1 def mapAffineInterp2D( iPtsRef, iPtsMov ):
2     # Tu napišite python kodo funkcije
3     return oMat2D

```

kjer je *iPtsRef* matrika s koordinatami referenčnih točk, *iPtsMov* pa matrika s koordinatami premičnih točk. Obe matriki sta dimenzij 3×2 . V funkciji lahko inverz 3×3 matrike izračunamo analitično ali z ukazom `numpy.linalg.inv()`. Funkcija vrne 2D afino preslikavo v obliki homogene matrike *oMat2D*, ki ima dimenzije 3×3 . Preverite delovanje funkcije tako, da **tri točke** izbrane oblike preslikate s poljubnimi parametri 2D afine preslikave in preverite ali funkcija vrne preslikavo v *oMat2D*, ki je enaka vaši 2D afini preslikavi. ■

```

1 def mapAffineInterp2D( iPtsRef, iPtsMov ):
2     """Afina interpolacijska poravnava"""
3     # pretvori v matrike
4     iPtsRef = np.asmatrix( iPtsRef )
5     iPtsMov = np.asmatrix( iPtsMov )
6     # po potrebi dodaj homogeno koordinato
7     if iPtsRef.shape[1] != 3:
8         iPtsRef = addHomo(iPtsRef)
9     if iPtsMov.shape[1] != 3:

```

```

10     iPtsMov = addHomo(iPtsMov)
11     # afina interpolacija
12     iPtsRef = iPtsRef.transpose()
13     iPtsMov = iPtsMov.transpose()
14     oMat2D = iPtsMov * np.linalg.pinv( iPtsRef )
15     return oMat2D
16
17 # preizkus funkcije
18 letala = np.load('letala.npy', allow_pickle=True)
19 let = np.matrix(letala[0])
20 # določi tri točke za poravnavo
21 idx = np.floor(rand(3) * let.shape[0]).astype('uint32')
22 iPtsRef = let[idx, :]
23 # preslikaj točke s poljubno afino preslikavo
24 oMat2D = transAffine2D( iScale=(1, 1), iTrans=(10, 22), iRot=10, iShear=(0.1,
25     0) )
26 iPtsMov = (oMat2D * addHomo(iPtsRef).transpose()).transpose()
27 # določi matriko preslikave z afino interpolacijo
28 oMat2D_t = mapAffineInterp2D( iPtsRef, iPtsMov )
29 # izpiši matriki in primerjavo
30 print('Afina preslikava:\n', oMat2D)
31 print('Določena z interpolacijsko poravnavo 3 parov točk:\n', oMat2D_t)
32 print('Razlika v elementih matrike:\n', oMat2D-oMat2D_t)

```

Funkcija vrne naslednji rezultat (lahko se malenkost razlikuje glede na naključno izbiro treh korespondenčnih točk v vrstici 21 gornje Python kode):

```

Afina preslikava:
[[ 1.00217257 -0.0751674  10.         ]
 [ 0.17364818  0.98480775 22.         ]
 [ 0.          0.          1.          ]]
Določena z interpolacijsko poravnavo 3 parov točk:
[[ 1.00217257 -0.0751674  10.         ]
 [ 0.17364818  0.98480775 22.         ]
 [ 0.          0.          1.          ]]
Razlika v elementih matrike:
[[ 6.66133815e-16 -1.38777878e-16  5.34683409e-13]
 [ 9.07607323e-15  1.44328993e-15 -3.77653464e-12]
 [ 0.00000000e+00  0.00000000e+00  8.88178420e-16]]

```

V splošnem so razlike v elementih matrike razlike zelo majhne, reda 10^{-16} , kar je pričakovan red numerične napake 32-bitnega zapisa z decimalno vejico.

Vaja 7.3 Napišite funkcijo, ki med pari korespondenčnih točk določi 2D afino aproksimacijsko poravnavo:

```

1 def mapAffineApprox2D( iPtsRef, iPtsMov ):
2     # Tu napišite python kodo funkcije
3     return oMat2D

```

kjer je $iPtsRef$ matrika s koordinatami referenčnih točk, $iPtsMov$ pa matrika s koordinatami premičnih točk. Obe matriki sta dimenzij $K \times 2$, kjer je $K \geq 3$. Funkcija vrne 2D afino preslikavo v obliki homogene matrike $oMat2D$, ki ima dimenzije 3×3 . Preverite delovanje funkcije tako,

da vse točke izbrane oblike preslikate s poljubnimi parametri 2D afine preslikave in preverite ali funkcija vrne preslikavo v oMat2D, ki je enaka vaši 2D afini preslikavi. ■

Funkcija `mapAffineApprox2D` implementira afino aproksimacijsko poravnavo med dvema množicama 2D točk z uporabo minimizacije po metodi najmanjših kvadratov. Algoritem najprej pretvori vhodne točke v matrično obliko in jim doda homogeno koordinato, če ta še ni prisotna. Nato reši sistem linearnih enačb z uporabo normalne enačbe $\mathbf{X} = \mathbf{B}\mathbf{A}^T(\mathbf{A}\mathbf{A}^T)^{-1}$, kjer $\mathbf{A}$ predstavlja referenčne točke in $\mathbf{B}$ premikajoče se točke. Ta pristop je matematično ekvivalenten izračunu psevdoinverza, vendar bolj numerično stabilen za slabo pogojene probleme. Rezultat je homogena 3×3 transformacijska matrika, ki minimizira vsoto kvadratov razdalj med transformiranimi referenčnimi točkami in ciljimi točkami. Metoda je še posebej primerna za primere, ko je število točk večje od minimalno potrebnega za določitev afine transformacije. Rešitev napišemo:

```
1 def mapAffineApprox2D( iPtsRef, iPtsMov ):
2     """Afina aproksimacijska poravnava"""
3     # pretvori v matrike
4     iPtsRef = np.asmatrix( iPtsRef ).astype('float')
5     iPtsMov = np.asmatrix( iPtsMov ).astype('float')
6     # po potrebi dodaj homogeno koordinato
7     if iPtsRef.shape[1]!=3:
8         iPtsRef = addHomo(iPtsRef)
9     if iPtsMov.shape[1]!=3:
10        iPtsMov = addHomo(iPtsMov)
11    # afina aproksimacija (s psevdoinverzom)
12    iPtsRef = iPtsRef.transpose()
13    iPtsMov = iPtsMov.transpose()
14    oMat2D = iPtsMov * iPtsRef.transpose() * np.linalg.inv( iPtsRef *
15        iPtsRef.transpose() )
16    return oMat2D
```

Delovanje funkcije preverimo na podoben način kot pri prejšnji vaji 7.2 in dobimo naslednji rezultat:

```
Afina preslikava:
[[ 1.00217257 -0.1503348  10.         ]
 [ 0.17364818  1.96961551  22.         ]
 [ 0.         0.         1.         ]]
Določena z aproksimacijsko poravnavo 3 parov točk:
[[ 1.00217257e+00 -1.50334805e-01  1.00000000e+01]
 [ 1.73648178e-01  1.96961551e+00  2.20000000e+01]
 [ 2.16840434e-19  0.00000000e+00  1.00000000e+00]]
Razlika v elementih matrike:
[[-2.88657986e-15  1.67088565e-14 -2.24531505e-12]
 [-1.42941214e-14 -4.19664303e-14  1.71418435e-11]
 [-2.16840434e-19  0.00000000e+00  1.77635684e-15]]
```

V splošnem so razlike v elementih matrike razlike zelo majhne, reda 10^{-16} , kar je pričakovan red numerične napake 32-bitnega zapisa z decimalno vejico.

Vaja 7.4 Napišite funkcijo, ki poišče pare korespondenčnih točk med dvema množicama točk:

```
1 def findCorrespondingPoints( iPtsRef, iPtsMov ):
2     # Tu napišite python kodo funkcije
```

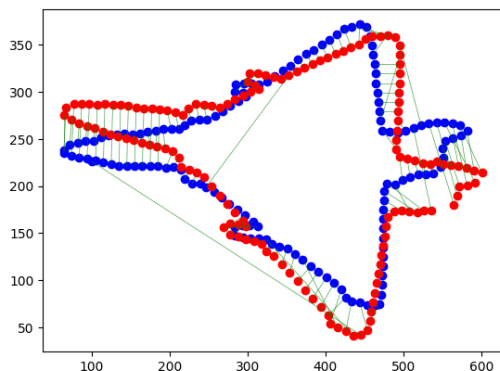
```
3     return iPtsRef_t, iPtsMov_t
```

kjer je `iPtsRef` matrika s koordinatami referenčnih točk, `iPtsMov` pa matrika s koordinatami premičnih točk. Matriki imata dimenziji $K_R \times 2$ in $K_M \times 2$. Funkcija vrne sortirane koordinate `iPtsRef_t` in `iPtsMov_t` tako da istoležeče vrstice predstavljajo pare korespondenčnih točk. ■

Funkcija najprej inicializira matriko razdalj, kjer izračuna evklidsko razdaljo med vsemi pari točk v množicah `iPtsRef` in `iPtsMov`. Nato iterativno izbira najbližje pare točk po principu najmanjše razdalje, pri čemer vsak izbran par izloči iz nadaljnjega iskanja, kar zagotavlja bijektivno preslikavo (tj. ena točka pripada natanko eni točki). Ta postopek se ponavlja, dokler niso vsi možni pari porabljeni. Funkcija na koncu vrne le tiste točke, za katere je bilo mogoče najti pripadajočo točko (uporabljamo tudi izraz *korespondenco*), kar omogoča robustno obravnavo primerov, ko množici točk nista enako veliki. Rešitev napišemo:

```
1 def findCorrespondingPoints( iPtsRef, iPtsMov ):
2     """Poišči korespondence kot najbližje točke"""
3     # inicializiraj polje indeksov
4     idxPair = -np.ones( (iPtsRef.shape[0],1) ).astype('int32')
5     idxDist = np.ones( (iPtsRef.shape[0],iPtsMov.shape[0]) )
6     for i in range(iPtsRef.shape[0]):
7         for j in range(iPtsMov.shape[0]):
8             idxDist[i,j] = np.sum((iPtsRef[i,:] - iPtsMov[j,:])**2)
9     # doloci bijektivno preslikavo
10    while not np.all(idxDist==np.inf):
11        i, j = np.where(idxDist==np.min(idxDist))
12        idxPair[i[0]] = j[0]
13        idxDist[i[0],:] = np.inf
14        idxDist[:,j[0]] = np.inf
15    # doloci pare točk
16    idxValid, idxNotValid = np.where(idxPair>=0)
17    idxValid = np.array( idxValid ).flatten()
18    iPtsRef_t = iPtsRef[idxValid,:]
19    iPtsMov_t = iPtsMov[idxPair[idxValid].flatten(),:]
20    return iPtsRef_t, iPtsMov_t
21
22 # preizkus funkcije
23 d = 10 # faktor decimacije točk
24 let = np.matrix(letala[0])
25 iPtsRef = let[:,d]
26 let = np.matrix(letala[12])
27 iPtsMov = let[:,d]
28 # klic funkcije
29 iPtsRef_t, iPtsMov_t = findCorrespondingPoints( iPtsRef, iPtsMov )
30
31 # izris oblik
32 plt.plot(iPtsRef_t[:,0], iPtsRef_t[:,1], 'ob')
33 plt.plot(iPtsMov_t[:,0], iPtsMov_t[:,1], 'or')
34
35 # izris povezav med pripadajočimi pari točk
36 for (ref_pt, mov_pt) in zip(iPtsRef_t, iPtsMov_t):
37     plt.plot([ref_pt[0], mov_pt[0]],
38             [ref_pt[1], mov_pt[1]],
39             'g-', linewidth=0.5, alpha=0.7)
```

Rezultat iskanja pripadajočih parov točk med dvema oblika prikazuje slika 7.3.



Slika 7.3: Prikaz oblike letala pri indeksu 0 (*modre pike*) in oblike letala pri indeksu 12 (*rdeče pike*) ter pripadajoči pari točk (*zeleno črte*).

Vaja 7.5 Napišite funkcijo za poravnavo dveh oblik z metodo iterativno najbližje točke (ICP):

```
1 def alignICP2D( iPtsRef, iPtsMov, iEps, iMaxIter ):
2     # Tu napišite python kodo funkcije
3     return oMat2D, oErr
```

kjer sta $iPtsRef$ in $iPtsMov$ matriki s koordinatami referenčnih in premičnih točk. Matriki imata dimenziji $K_R \times 2$ in $K_M \times 2$. $iEps$ in $iMaxIter$ določata pogoje za zaustavitev postopka, kjer je $iEps$ najmanjša sprememba parametrov preslikave, $iMaxIter$ pa maksimalno število korakov k_{max} . V funkciji uporabite 2D afino aproksimacijsko poravnavo, funkcija pa naj vrne optimalno 2D afino preslikavo med referenčnimi in premičnimi točkami v obliki homogene matrike $oMat2D$ in vektor $oErr$ velikosti $k_{max} \times 1$, ki podaja napako Σ^2 med korespondenčnimi točkami v vsakem koraku metode ICP.

Preverite pravilnost delovanja funkcije tako, da vse točke izbrane oblike preslikate s poljubnimi parametri 2D afine preslikave in preverite ali funkcija vrne preslikavo v $oMat2D$, ki je podobna dani vaši 2D afini preslikavi in da je končna napaka $oErr$ manjša od začetne napake in približno enaka 0. Naložite referenčne in premične točke dveh poljubnih, a različnih oblik iz dane zbirke in preverite, da funkcija obliki ustrezno poravnava. ■

Algoritem ICP iterativno izvaja tri ključne korake: (1) iskanje korespondenčnih parov točk s funkcijo `findCorrespondingPoints`, (2) izračun optimalne afine preslikave s funkcijo `mapAffineApprox2D` in (3) aplikacija preslikave na premikajoče se točke. Postopek se ponavlja dokler napaka poravnave ne pade pod prag $iEps$ ali dokler ne dosežemo največjega števila iteracij $iMaxIter$. V vsaki iteraciji algoritem beleži napako poravnave in akumulira transformacijske matrike. Končna transformacija je kompozitum vseh delnih transformacij, kar omogoča postopno izboljšanje poravnave. Rešitev lahko napišemo:

```
1 def alignICP( iPtsRef, iPtsMov, iEps=1e-3, iMaxIter=50 ):
2     """Postopek iterativno najblizje tocke"""
3     # pretvori v polje
4     iPtsRef = np.asarray( iPtsRef ).astype('float')
5     iPtsMov = np.asarray( iPtsMov ).astype('float')
6     iPtsRef = iPtsRef[:, :2]
7     iPtsMov = iPtsMov[:, :2]
8     # inicializiraj izhodne parametre
```

```

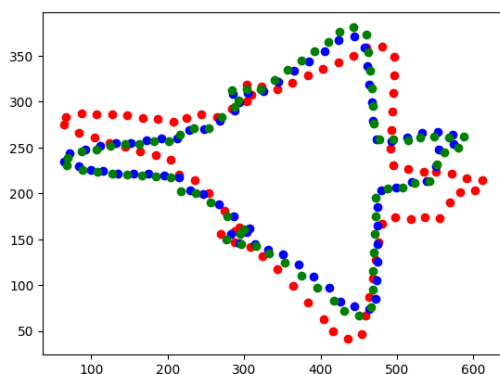
9   curMat = []; oErr = []; iCurIter = 0
10  # zacni iterativni postopek
11  while True:
12      # poisci korespondencne pare tock
13      iPtsRef_t, iPtsMov_t = findCorrespondingPoints( iPtsRef, iPtsMov )
14      # izracunaj napako
15      oErr.append( np.sqrt(np.sum( (iPtsRef_t - iPtsMov_t)**2)) /
16                  iPtsRef_t.shape[0])
17      # doloci afino aproksimacijsko preslikavo
18      oMat2D = mapAffineApprox2D( iPtsRef_t, iPtsMov_t, iUsePseudoInv=True )
19      oMat2D = np.linalg.inv(oMat2D)
20      curMat.append( oMat2D )
21      # posodobi premicne tocke
22      iPtsMov = (oMat2D * addHomo(iPtsMov).transpose()).transpose()
23      iPtsMov = np.asarray( iPtsMov[:,2] )
24      # preveri kontrolne parametre
25      iCurIter = iCurIter + 1
26      dMat = np.abs(oMat2D - transAffine2D())
27      if iCurIter>iMaxIter or np.all(dMat<iEps):
28          break
29      print('iter = ' + str(iCurIter) + ', err = ' + str(oErr[-1]))
30      # doloci kompozitum preslikav
31      oMat = transAffine2D()
32      for i in range(len(curMat)):
33          oMat = curMat[i] * oMat
34      return oMat, oErr
35
36 # preizkus funkcije
37 d = 20 # faktor decimacije
38 letala = np.load('letala.npy', allow_pickle=True)
39 let = np.matrix(letala[0])
40 iPtsRef = let[:,d] # referenčna oblika
41 let = np.matrix(letala[12])
42 iPtsMov = let[:,d] # premična oblika
43 # določi matriko preslikave
44 oMat2D_t, oErr = alignICP( iPtsRef, iPtsMov, iEps=1e-12, iMaxIter=50,
45                             iVerbose=1 )
46 # preslikaj premične točke
47 iPtsMov_t = (oMat2D_t * addHomo(iPtsMov).transpose()).transpose()
48 # nariši referenčne, premične in preslikane premične točke
49 plt.plot(iPtsMov[:,0],iPtsMov[:,1], 'or')
50 plt.plot(iPtsRef[:,0],iPtsRef[:,1], 'ob')
51 plt.plot(iPtsMov_t[:,0],iPtsMov_t[:,1], 'og')

```

Preizkus funkcije smo opravili na dveh različnih oblikah letal, tj. oblik letal pri indeksih 0 in 12. Rezultat ICP poravnave premične oblike na referenčno prikazuje slika 7.4, iz katere je razvidno dobro ujemanje referenčne in premične oblike (*rdeče in zelene točke*) po poravnavi z ICP algoritmom.

7.4 Naloge in vprašanja

Poročilo naloge pripravite v obliki Python skripte, ki naj izvede zahtevane izračune oz. klice funkcij in izriše pripadajoče slike oz. grafe. Vaša dolžnost je, da v skripti uvozite ustrezne Python knjižnice in dodate morebitne podporne skripte. Skripta se mora izvesti in poustvariti



Slika 7.4: Prikaz oblike letala pri indeksu 0 (*modre pike*) in oblike letala pri indeksu 12 (*rdeče pike*) ter pripadajoči pari točk (*zelene pike*).

rezultate, kot se zahtevajo v nalogah. Na vprašanja odgovorite v obliki komentarja, naprimer `# Naloga X.2: To je moj odgovor..`

Naloga 7.1 Izvedite poravnavo vseh 2D oblik z indeksi [1, 29] v datoteki `letala.npy` na obliko z indeksom 0 z metodo iterativno najbližje točke. Izrišite v eno sliko vse 2D oblike v začetni legi, v drugo sliko pa vse oblike v poravnani legi.

Za vsako poravnavo izračunajte napako pred poravnavo in po njej. V eno sliko narišite histogram vseh napak v začetni legi, v drugo sliko pa histogram napak v poravnani legi. Izpišite vrednost napake in indeks oblike, pri kateri dobite največjo in najmanjšo napako v začetni legi. Oblike tudi izrišite, zraven pa z drugo barvo dorišite še referenčno obliko. Enako naredite tudi za oblike po poravnavi.

Naloga 7.2 Naložite poljubno obliko iz dane zbirke in načrtajte ustrezno zaporedje 2D linearnih preslikav tako, da se bo oblika rotirala okoli svojega središča. Prikažite delovanje preslikave tako, da obliko rotirate od $0 - 360^\circ$ s korakom 5° ter hkrati prikazujete točke oblike v istem prikaznem oknu. Za vsak korak z metodo iterativno najbližje točke poravnajte rotirano obliko v nerotirano obliko (kot 0°). Ali je poravnava uspešna za vse rotacije? Pokažite tako, da za vsak korak izračunate napako kot povprečno razdaljo med istoindeksiranimi točkami v referenčni in poravnani obliki. Napako prikažite na grafu v odvisnosti od kota rotacije.

Naloga 7.3 Toga poravnava, ki se pogosto uporablja v praksi, je vedno aproksimacijska, saj zahtevano minimalno število parov korespondenčnih točk vodi do predoločenega sistema enačb – dobimo namreč več enačb kot pa je neznanih parametrov. Za poravnavo 2D oblik potrebujemo $K \geq 2$ korespondenčnih točk, preslikavo pa določimo z minimizacijo povprečne kvadratno Evklidske razdalje med točkami:

$$\Sigma^2 = \frac{1}{K} \sum_{i=1}^K [(x_i \cos \alpha - y_i \sin \alpha + t_x - u_i)^2 + (x_i \sin \alpha + y_i \cos \alpha + t_y - v_i)^2],$$

ki jo odvajamo po parametrih t_x , t_y in α in odvode postavimo na nič. Rešitev sistema enačb je:

$$\alpha = -\operatorname{atan} \frac{\overline{u}y - \overline{v}x - \overline{u}\bar{y} + \overline{v}\bar{x}}{\overline{u}x + \overline{v}y - \overline{u}\bar{x} - \overline{v}\bar{y}},$$

$$t_x = \overline{u} - \bar{x} \cos \alpha + \bar{y} \sin \alpha,$$

$$t_y = \overline{v} - \bar{x} \sin \alpha - \bar{y} \cos \alpha.$$

Napišite funkcijo, ki med pari korespondenčnih točk določi 2D togo aproksimacijsko poravnavo:

```

1 def mapRigid2D( iPtsRef, iPtsMov ):
2     # Tu napišite python kodo funkcije
3     return oMat2D

```

kjer sta `iPtsRef` in `iPtsMov` matriki s koordinatami referenčnih in premičnih točk. Obe matriki imata dimenzije $K \times 2$, pri čemer je $K \geq 2$. Funkcija naj vrne 2D togo preslikavo v obliki homogene matrike `oMat2D` z dimenzijami 3×3 . Prikažite delovanje funkcije tako, da **vse točke** izbrane oblike preslikate s poljubnimi parametri 2D toge preslikave in preverite ali funkcija vrne preslikavo v `oMat2D`, ki je enaka vaši 2D togi preslikavi.

Naloga 7.4 Izvedite poravnavo vseh 2D oblik z indeksi [1, 29] v datoteki `letala.npy` na obliko z indeksom 0 z metodo iterativno najbližje točke, pri tem pa uporabite **2D togo aproksimacijsko poravnavo**. Izrišite v eno sliko vse 2D oblike v začetni legi, v drugo sliko pa vse oblike v poravnani legi. Primerjajte dobljeni rezultat z rezultatom, ki ste ga dobili pri nalogi 7.1.

Naloga 7.5 Napišite funkcijo, ki ustvari 4×4 matriko poljubne 3D affine preslikave:

```

1 def transAffine3D( iScale, iTrans, iRot, iShear ):
2     # Tu napišite python kodo funkcije
3     return oMat3D

```

kjer je `iScale` trivrstični vektor s parametri skale k_x, k_y in k_z , `iTrans` trivrstični vektor s parametri translacije t_x, t_y in t_z , `iRot` trivrstični vektor s koti rotacije α, β in γ v stopinjah in `iShear` šestvrstični vektor s parametri striga $g_{xy}, g_{xz}, g_{yx}, g_{zx}, g_{yz}, g_{zy}$. Funkcija vrne homogeno matriko `oMat3D`, ki ima dimenzije 4×4 . Preverite delovanje funkcije na 3D oblikah v datoteki `glave.npy`² tako, da točke izbrane oblike preslikate s poljubnimi parametri striga, rotacije, translacije in skaliranja, nato pa še s poljubno 3D afino preslikavo. Vhodne in preslikane točke oblike prikažite z dano funkcijo `renderSurface.py`.

Izbrano obliko rotirajte za 180° okrog svojega središča in okrog poljubne osi. Prikažite glavo pred in po poravnavi.

Naloga 7.6 Napišite funkcijo za poravnavo dveh 3D oblik z metodo iterativno najbližje točke (ICP):

```

1 def alignICP3D( iPtsRef, iPtsMov, iEps, iMaxIter ):
2     # Tu napišite python kodo funkcije
3     return oMat3D, oErr

```

kjer sta `iPtsRef` in `iPtsMov` matriki s koordinatami referenčnih in premičnih točk. Matriki imata dimenziji $K_R \times 3$ in $K_M \times 3$. `iEps` in `iMaxIter` določata pogoje za zaustavitev postopka, kjer je `iEps` najmanjša sprememba parametrov preslikave, `iMaxIter` pa maksimalno število korakov k_{max} . V funkciji uporabite 3D afino aproksimacijsko poravnavo, funkcija pa naj vrne optimalno 3D afino preslikavo med referenčnimi in premičnimi točkami v obliki homogene matrike `oMat3D` in vektor `oErr`, ki podaja napako Σ^2 med korespondenčnimi točkami v vsakem koraku metode ICP.

Preverite pravilnost delovanja funkcije tako, da vse točke izbrane oblike preslikate s poljubnimi parametri 3D affine preslikave in preverite ali funkcija vrne preslikavo v `oMat3D`, ki je podobna dani vaši 3D afini preslikavi in da je končna napaka `oErr` manjša od začetne napake in približno enaka 0. Naložite referenčne in premične točke dveh poljubnih, a različnih oblik iz dane zbirke glav in preverite, da funkcija obliki ustrezno poravna.

²Vseh pet 3D oblik naložite z ukazom `glave = numpy.load('glave.npy')` in so zapisane v seznamu tipa `list`, točke prve oblike pa dobite z ukazom `x=glave[0]`.

Izvedite poravnavo vseh 3D oblik [1,4] na obliko z indeksom 0. Izpišite najmanjšo in največjo vrednost napake `oErr` pred poravnavno in po njej.

Naloga 7.7 Na podlagi enačb v dodatku B implementirajte funkcijo za 3D togo aproksimacijsko poravnavo korespondenčnih parov točk. Preizkusite delovanje v funkciji `alignICP3D( iPtsRef, iPtsMov, iEps, iMaxIter )`, pri čemer razširite funkcijo tako, da bo z vhodnim parametrom `iTransformType` možna izbira preslikave, in sicer `'rigid'` ali `'affine'`. Izvedite poravnavo vseh 3D oblik [1,4] na obliko z indeksom 0. Izpišite najmanjšo in največjo vrednost napake `oErr` pred poravnavno in po njej. Katera preslikava vrne manjšo napako?

Naloga 7.8 V pythonu naložite knjižnico `pycpd`³, ki vsebuje poravnavo z metodo koherentnega premika točk (CPD; *angl.* coherent point drift) [14]. Izvedite poravnavo vseh 2D oblik [1,29] v datoteki `letala.npy` na obliko z indeksom 0 z metodo CPD, enkrat z afino poravnavo (`affine_registration`), drugič pa še s togo (`rigid_registration`). Za vsako od poravnav izrišite v eno sliko vse 2D oblike v poravnani legi. Za obe poravnavi izračunajte porazdelitev napake po poravnavi oblik in ju primerjajte s porazdelitvijo, ki jo dobite pri metodi ICP. Napako izračunajte kot povprečno razdaljo med korespondenčnimi točkami.

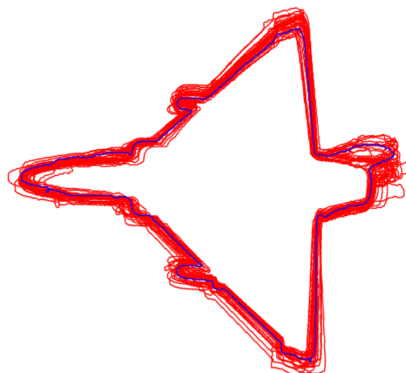
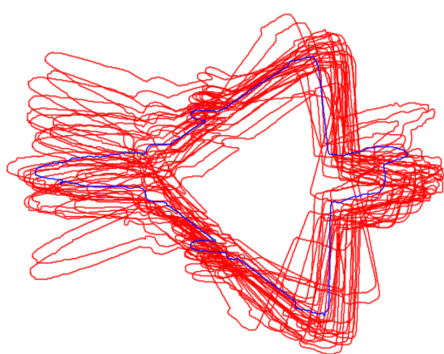
³<https://github.com/siavashk/pycpd>

Rezultati nekaterih nalog:

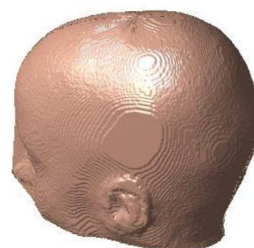
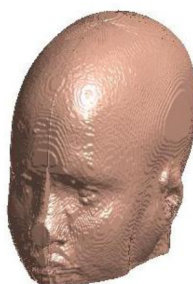
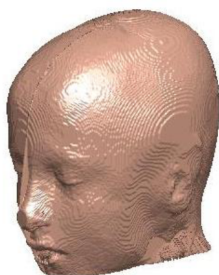
Preslikava 2D oblik na referenčno obliko (*modra črta*)

Brez poravnave

Z afino aproksimacijo



Poljubne afine preslikave 3D oblike



Aplikacije robotskega vida

8 Geometrijska kalibracija kamere ... 113

- 8.1 Geometrijski kalibri
- 8.2 Model preslikave prostor-kamera
- 8.3 Prileganje modela s slikami kalibra
- 8.4 Vaje z rešitvami
- 8.5 Naloge in vprašanja

9 Določanje poze objektov v prostoru . 127

- 9.1 Projekcija iz 3D v 2D
- 9.2 Mera podobnosti
- 9.3 Optimizacijski postopek
- 9.4 Vaje z rešitvami
- 9.5 Naloge in vprašanja

10 Rekonstrukcija 3D oblik 149

- 10.1 Fotometrični stereo in Lambertov model
- 10.2 Izračun albeda in normal na površino objekta
- 10.3 Rekonstrukcija 3D oblike
- 10.4 Vaje z rešitvami
- 10.5 Naloge in vprašanja

11 Sledenje in analiza gibanja 161

- 11.1 Parametrizacija preslikave
- 11.2 Linearizacija nelinearne kriterijske funkcije
- 11.3 Zaprto-zančna posodobitev preslikave
- 11.4 Piramidna implementacija
- 11.5 Vaje z rešitvami
- 11.6 Naloge in vprašanja

12 Lokalizacija in mapiranje okolja 175

- 12.1 Algoritem stereo vida
- 12.2 Lokalizacija in mapiranje okolja
- 12.3 Vaje z rešitvami
- 12.4 Naloge in vprašanja

13 Vizualna kontrola kakovosti 195

- 13.1 Sistem za razpoznavanje vzorcev
- 13.2 Vrednotenje sistemov odločanja
- 13.3 Vaje z rešitvami
- 13.4 Naloge in vprašanja



8. Geometrijska kalibracija kamere

S postopkom **geometrijske kalibracije** slikovnega sistema zagotovimo enakomerno absolutno velikost slikovnega elementa, ki je sicer v splošnem lahko odvisna od položaja v sliki¹ in dodatno popačena zaradi optičnih aberacij, med katerimi so najpogostejše radialne distorzije tipa sodček in blazinica. Po geometrijski kalibraciji slikovnega sistema in preslikavi slike v metrični prostor lahko izvajamo meritve dimenzij, ploščin in volumnov objektov na slikah v absolutnih metričnih enotah (npr. mm, mm², mm³).

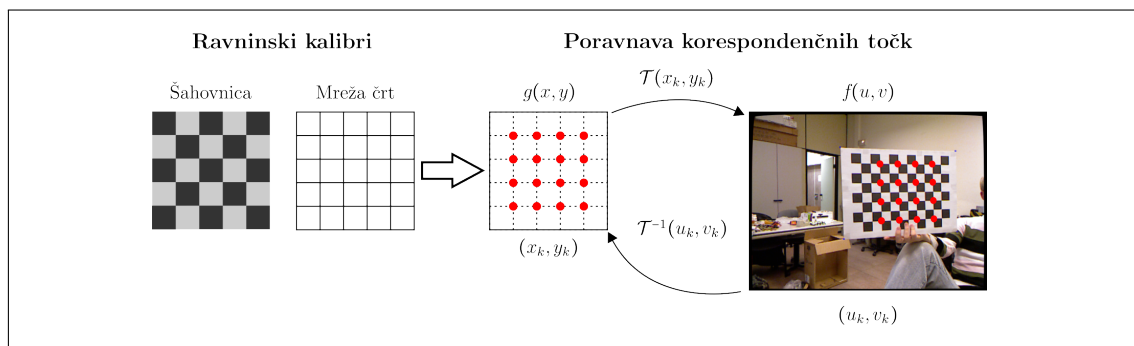
8.1 Geometrijski kalibri

Za geometrijsko kalibracijo 2D slikovnih sistemov **uporabljamo ravninske kalibre** z enostavnimi, a dobro definiranimi vzorci, kot so **šahovnica, mreža tankih črt ali diskretno polje točk**. Geometrijska kalibracija vključuje poravnavo vzorcev zajete $f(u, v)$ in referenčne $g(x, y)$ slike kalibracijskega objekta. Uporabimo lahko celotno slikovno informacijo ali pa izluščimo korespondenčne pare točk referenčne in kalibracijske slike ter z njimi določimo parametre preslikave oz. poravnavo tako, da se preslikane točke referenčne slike čim bolj prekrivajo s korespondenčnimi točkami kalibracijske slike in obratno, kot prikazuje slika 8.1.

8.2 Model preslikave prostor-kamera

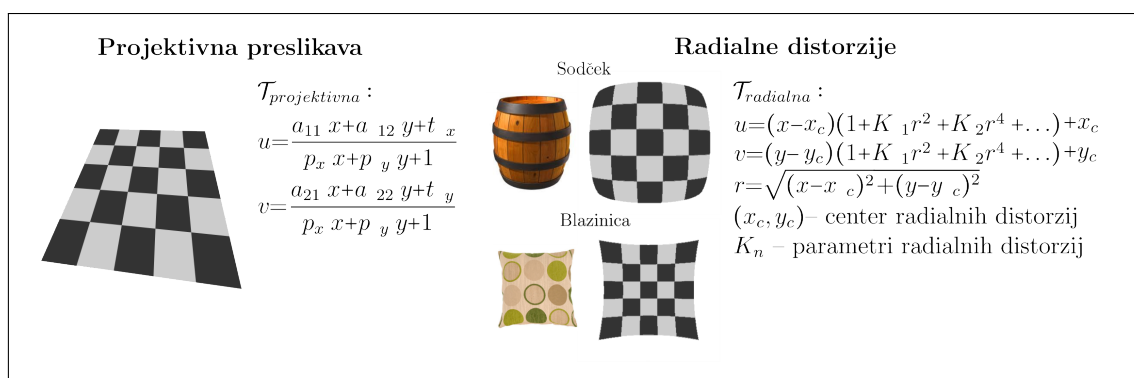
V praksi se za geometrijsko kalibracijo slikovnih sistemov običajno uporablja **projektivna geometrijska preslikava**, ki je za preslikavo med dvema ravninama določena z 8 parametri ($a_{11}, a_{12}, a_{21}, a_{22}, t_x, t_y, p_x, p_y$) in ki jo zapišemo z nehomogeno 3×3 matriko, za odpravo popačenj tipa sodček in blazinica pa **Brownov model radialnih distorzij**, ki je določen s centrom (u_c, v_c) in utežmi radialnih funkcij K_i . Enačbe in vpliv omenjenih preslikav je prikazan na sliki 8.2. Geometrijsko kalibracijo izvedemo tako, da točke v metričnem prostoru s projektivno preslikavo preslikamo v prostor slike in jih nato še popačimo z modelom radialnih distorzij. Ker je Brownov model radialnih distorzij nelinearen, je za določanje parametrov modela radialnih distorzij potrebno uporabiti op-

¹npr. zaradi vpliva projekcije



Slika 8.1: Kalibracija kamere z ravninskimi kalibri z visoko-kontrastnimi vzorci na osnovi zaznave in poravnave korespondenčnih oz. pripadajočih parov točk.

timizacijski postopek, ki minimizira kriterijsko funkcijo, npr. srednjo kvadratno Evklidsko razdaljo $E^2 = \sum_{k=1}^N \|\mathcal{T}_{radialna}(\mathcal{T}_{projektivna}(x_k, y_k)) - (u_k, v_k)\|^2$ med korespondenčnimi točkami na kalibru.



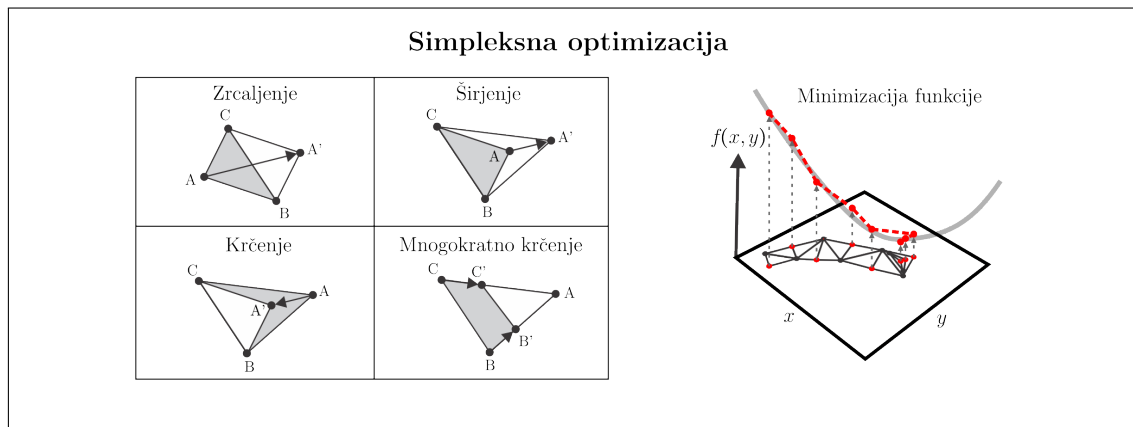
Slika 8.2: Model geometrijske preslikave za standardne optične sisteme s kamero je zaporedn projekтивna preslikava in radialna distorzija.

8.3 Prileganje modela s slikami kalibra

Za določanje optimalnih parametrov modela radialnih distorzij in 2D projektivne preslikave boste uporabili postopek **Nelder-Mead simpleksne optimizacije**. S simpleksno optimizacijo lahko poiščemo lokalni optimum poljubne funkcije več spremenljivk. Simpleks predstavlja geometrijsko strukturo, s pomočjo katere vzorčimo parametrični prostor in je za funkcije dveh spremenljivk $f(x, y)$ predstavljen s tremi oglišči oz. s trikotnikom. S simpleksno optimizacijo poljubno funkcijo minimiziramo tako, da oglišča z največjo vrednostjo $f(x, y)$ nadomestimo z novimi oglišči. Nova oglišča lahko določimo na štiri načine: 1) zrcaljenje, 2) širjenje, 3) krčenje in 4) mnogokratno krčenje (slika 8.3). Ko oglišča nadomestimo dobimo nov simpleks in poiščemo pripadajoče vrednosti $f(x, y)$ v novih ogliščih. Postopek ponavljamo in ga ustavimo, dokler velikost simpleksa ne pade pod neko minimalno vrednost, položaj lokalnega optima pa predstavlja oglišče z optimalno vrednostjo $f(x, y)$.

8.4 Vaje z rešitvami

Pri vaji boste napisali funkcije za popravek radialnih distorzij in 2D projektivno preslikavo. Z določanjem oglišč na sliki kalibracijskega objekta `calibration-object.jpg` in njihovo preslikavo v metrični prostor s poravnavo korespondenčnih oglišč boste izvedli geometrijsko kalibracijo



Slika 8.3: Model geometrijske preslikave za standardne optične sisteme s kamero je zaporedn projekтивna preslikava in radialna distorzija.

slikovnega sistema. Nato boste sliko kljunastega merila `test-object.jpg`, ki je bila zajeta z istim slikovnim sistemom kot slika kalibra, preslikali v metrični prostor in z merjenjem dolžine na kljunastem merilu preverili ali je geometrijska kalibracija pravilna. Koda vaje je dostopna na

[Git repozitoriju](#).

Najprej uvozimo Python knjižnice in naložimo ter prikažemo sliki za izvedbo vaje:

```
1 import scipy.ndimage as ni
2 import matplotlib.pyplot as plt
3 import matplotlib.cm as cm
4 import numpy as np
5 import scipy as sp
6 from scipy.interpolate import interpn
7 import PIL.Image as im
8
9 from rvlib import *
10
11 iCalImage = loadImage('calibration-object.jpg')
12 iTestImage = loadImage('test-object.jpg')
13
14 showImage( iCalImage, 'Kalibracijski objekt' )
15 showImage( iTestImage, 'Testni objekt' )
```

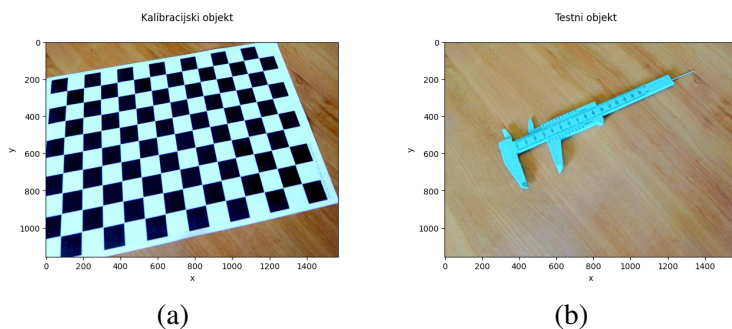
Zajeti sliki sta prikazani na sliki 8.4.

Vaja 8.1 Napišite funkcijo, ki preslika vhodne nepopačene koordinate (u, v) v spremenljivkah `iCoorU` in `iCoorV` z Brownovim modelom radialnih distorzij poljubnega reda n :

```
1 def transRadial( iK, iUc, iVc, iCoorU, iCoorV ):
2     # Tu napišite python kodo funkcije
3     return oCoorUd, oCoorVd
```

kjer je `iK` vrstični vektor $1 \times n$ parametrov radialnih distorzij, `iUc` in `iVc` pa predstavljata (u_c, v_c) koordinati centra radialnih distorzij. Funkcija vrne preslikane *popačene* koordinate točk v spremenljivkah `iCoorUd` in `iCoorVd`.

S pomočjo funkcije `numpy.meshgrid()` ustvarite diskretno mrežo koordinatnih točk in jih



Slika 8.4: Za vajo smo z istim slikovnim sistemom zajeli (a) sliko šahovnice kot kalibracijskega objekta in (b) sliko kljunastega merila.

preslikajte s poljubnim Brownovim modelom radialnih distorzij prvega reda ($n = 1$) ter narišite vhodne (u, v) in preslikane točke (u_d, v_d). Preverite vpliv položaja centra radialnih distorzij na preslikavo koordinatnih točk. Katere vrednosti iK predstavljajo distorzije tipa sodček in katere tipa blazinica? Pri katerih vrednostih parametrov bo ta preslikava identiteta? ■

Funkcija `transRadial` implementira Brownov model radialnih distorzij za popačenje slikovnih koordinat, ki se uporablja v fotogrametriji in računalniškem vidu za modeliranje geometrijskih napak objektiv. Najprej funkcija centrirá vhodne koordinate glede na optični center (iUc, iVc) in jih normalizira z deljenjem z največjim odklonom. Nato izračuna radialno razdaljo $R^2 = u_d^2 + v_d^2$ in uporabi polinom radialnih distorzij ($1 + \sum_{i=1}^n \kappa_i R^{2i}$), kjer so κ_i koeficienti distorzij iz vhodnega vektorja iK . Popačene koordinate se nato denormalizirajo in premaknejo nazaj v originalni koordinatni sistem. Rešitev zapišemo:

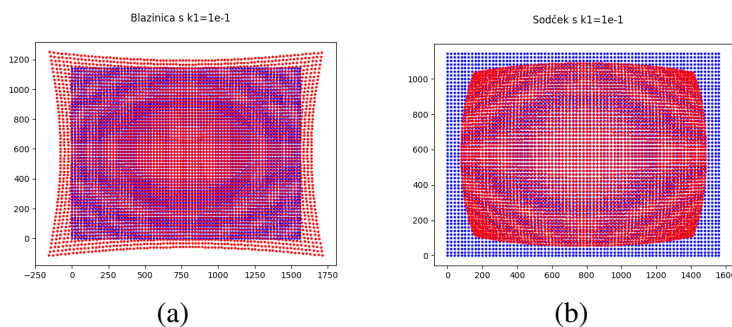
```
1 def transRadial( iK, iUc, iVc, iCoorU, iCoorV ):
2     """Funkcija za preslikavo z Brownovim modelom distorzij"""
3     # preveri vhodne podatke
4     iK = np.array( iK )
5     iCoorU = np.array( iCoorU )
6     iCoorV = np.array( iCoorV )
7     if np.size(iCoorU) != np.size(iCoorV):
8         print('Število U in V koordinat je različno!')
9     # odštej koordinate centra
10    oCoorUd = iCoorU - iUc; oCoorVd = iCoorV - iVc
11    # pripravi izhodne koordinate
12    sUd = np.max( np.abs( oCoorUd ) )
13    sVd = np.max( np.abs( oCoorVd ) )
14    oCoorUd = oCoorUd / sUd
15    oCoorVd = oCoorVd / sVd
16    # preračunaj radialna popačenja
17    R2 = oCoorUd**2.0 + oCoorVd**2.0
18    iK = iK.flatten()
19    oCoorRd = np.ones_like( oCoorUd )
20    for i in range(iK.size):
21        oCoorRd = oCoorRd + iK[i]*(R2**(i+1))
22    # izračunaj izhodne koordinate
23    oCoorUd = oCoorUd * oCoorRd * sUd + iUc
24    oCoorVd = oCoorVd * oCoorRd * sVd + iVc
25    return oCoorUd, oCoorVd
26
27 # preizkus funkcije:
```

```

28 # ustvari mrežo vzročnih točk slike
29 iCoorU, iCoorV = np.meshgrid(range(iCalImage.shape[1]),
30                               range(iCalImage.shape[0]),\
31                               sparse=False, indexing='xy')
32 # preslikaj točke (iK = -1e-1 in +1e-1, sodček in blazinica)
33 oCoorUd, oCoorVd = transRadial([1e-1], iCalImage.shape[1]/2.0,
34                                 iCalImage.shape[0]/2.0,\
35                                 iCoorU, iCoorV)
36
37 # prikaži originalne in preslikane točke
38 plt.close( 'all' )
39 s = 20
40 plt.plot(iCoorU[:,s,:].flatten(), iCoorV[:,s,:].flatten(),
41          'ob', markersize=2.0)
42 plt.plot(oCoorUd[:,s,:].flatten(), oCoorVd[:,s,:].flatten(),
43          'or', markersize=2.0)

```

Rezultat funkcije so popačene koordinate (oCoorUd, oCoorVd), ki upoštevajo nelinearne deformacije, značilne za širokokotne objektivne in objektivne ribje oko (ang. *fish-eye*). Model omogoča popraviljanje različnih vrst distorzij, vključno z blagimi barvnimi popačenjimi in ekstremnimi popačenji tipa ribje oko. Obliko in stopnjo popačenja lahko opazimo s preslikavo originalnih vzorčnih točk zajetih slik s poljubno izbranimi vrednostmi κ_i , kot prikazuje slika 8.5.



Slika 8.5: Vzorčne točke zajetih slik z (*modre točke*) in s funkcijo `transRadial()` preslikanih *rdečih točk*, za radialne distorzije tipa (a) blazinica ($\kappa_1 = 0.1$) in (b) sodček ($\kappa_1 = -0.1$).

Vaja 8.2 Napišite funkcijo, ki vhodne koordinate (x, y) v spremenljivkah `iCoorX` in `iCoorY` preslika s poljubno 2D projektivno preslikavo:

```

1 def transProjective2D( iPar, iCoorX, iCoorY ):
2     # Tu napišite python kodo funkcije
3     return oCoorU, oCoorV

```

kjer je `iPar` vrstični 1×8 vektor s parametri $(a_{11}, a_{12}, t_x, a_{21}, a_{22}, t_y, p_x, p_y)$. Funkcija vrne preslikane koordinate točk v spremenljivkah `oCoorU` in `oCoorV`.

Preverite delovanje funkcije tako, da vzorčne točke slike `test-object.jpg` preslikate s poljubnimi parametri projektivne preslikave in s funkcijo za interpolacijo prvega reda `scipy.interpolate.interpn()` v preslikanih točkah poiščete nove sivinske vrednosti. Prikažite preslikano sliko. ■

Funkcija `transProjective2D` implementira 2D projektivno preslikavo (homografijo), ki

omogoča transformacijo koordinat med dvema ravninama z uporabo 8-parametrskega modela. Preslikava uporablja projektivno geometrijo, kjer se vhodne koordinate (x,y) preslikajo v izhodne (u,v) po enačbah:

$$u = \frac{a_{11}x + a_{12}y + t_x}{p_x x + p_y y + 1}, \quad v = \frac{a_{21}x + a_{22}y + t_y}{p_x x + p_y y + 1} \quad (8.1)$$

kjer parametri predstavljajo elemente vektorja $iPar$. Ključna značilnost te preslikave je ohranjanje ravnih črt, vendar s spremembo njihovega relativnega razmerja in lege, kar omogoča modeliranje perspektivnih deformacij. Rešitev zapišemo:

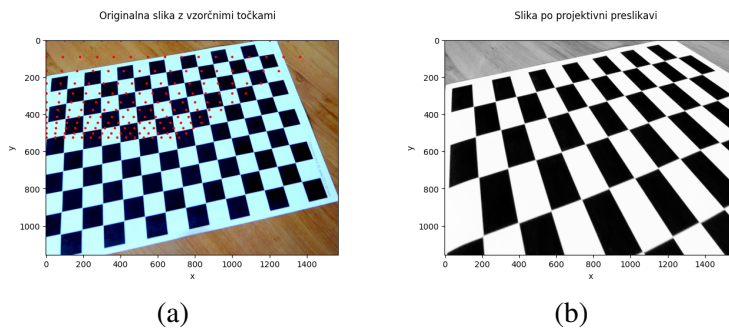
```

1 def transProjective2D( iPar, iCoorX, iCoorY ):
2     """Funkcija za projektivno preslikavo"""
3     # preveri vhodne podatke
4     iPar = np.asarray( iPar )
5     iCoorX = np.asarray( iCoorX )
6     iCoorY = np.asarray( iCoorY )
7     if np.size(iCoorX) != np.size(iCoorY):
8         print('Število X in Y koordinat je različno!')
9     # izvedi projektivno preslikavo
10    oDenom = iPar[6] * iCoorX + iPar[7] * iCoorY + 1
11    oCoorU = iPar[0] * iCoorX + iPar[1] * iCoorY + iPar[2]
12    oCoorV = iPar[3] * iCoorX + iPar[4] * iCoorY + iPar[5]
13    # vrni preslikane točke
14    return oCoorU/oDenom, oCoorV/oDenom
15
16 # preizkus funkcije:
17 # ustvari diskretno mrežo točk
18 iCoorX, iCoorY = np.meshgrid(range(iCalImage.shape[1]),
19                                range(iCalImage.shape[0]),\
20                                sparse=False, indexing='xy')
21 # preslikaj točke
22 iPar = [1, 0, 0, 0, 1, 0, 0, 0.001];
23 oCoorU, oCoorV = transProjective2D( iPar, iCoorX, iCoorY )
24
25 # interpolacija slike 1. reda (bilinearna)
26 iCalImageG = colorToGray( iCalImage )
27 s = 1; dy,dx = iCalImageG.shape
28 oInterpImageG = interpn((np.arange(dy), np.arange(dx)), iCalImageG,
29                          (oCoorV[:,s,:s],oCoorU[:,s,:s]), method = "linear", bounds_error=False) #
30                          interpolacija 0. reda
31
32 # prikaz slike z vzročnimi točkami
33 plt.imshow(iCalImage, cmap = cm.Greys_r) # prikazi sliko v novem oknu
34 plt.suptitle('Originalna slika z vzročnimi točkami') # nastavi naslov slike
35 plt.xlabel('x'); plt.ylabel('y')
36 s = 100; plt.plot(oCoorU[:,s,:s].flatten(),oCoorV[:,s,:s].flatten(),
37                  'or',markersize=2.0)
38
39 # prikaz prevzorčene (interpolirane) slike
40 showImage( oInterpImageG, 'Slika po projektivni preslikavi' )

```

Funkcija uvodoma preveri konsistentnost vhodnih podatkov, nato izvede matrično množenje v homogenih koordinatah in končno deli z normalizacijskim faktorjem $oDenom$ za pretvorbo nazaj v

kartezične koordinate. Takšne preslikave so široko uporabljene v računalniškem vidu za poravnavo slik, panoramsko zlivanje in 3D rekonstrukcijo. Primer projektivne preslikave prikazuje slika 8.6.



Slika 8.6: Prikaz (a) originalne slike z vzorčnimi točkami (*rdeče*) za projektivno preslikavo in (b) z interpolacijo 1. reda (bilinearna) prevzorčena slika.

Vaja 8.3 Preizkusite delovanje simpleksne optimizacije s pomočjo Pythonove funkcije `scipy.optimize.fmin()` tako, da poiščete minimum Rosenbrockove funkcije $f(x,y) = 100(y - x^2)^2 + (1 - x)^2$, ki je standardni testni primer za nelinearne optimizacijske algoritme. Pri tem pa uporabite začetni približek $x_0 = (1.2, 1)$.

Z nastavljanjem dodatnih parametrov v klicu funkcije `fmin(func, x0, ...)` nastavite prikaz informacij o iteracijah^a, maksimalno število iteracij^b, najmanjšo napako funkcije^c in najmanjši korak^d. Preverite, da je minimum $f(x,y)$ v točki $(1,1)$ in opazujte zaporedje operacij simpleksa. ■

^a`fmin(..., disp = 1, callback = fname)`, kjer je `fname(X)` funkcija za oblikovanje izpisa
^b`fmin(..., maxiter = 100)`
^c`fmin(..., ftol = 1e-6)`
^d`fmin(..., xtol = 1e-6)`

V kodi implementiramo kot funkcijo izraz $F(x,y) = 100(y - x^2)^2 + (1 - x)^2$. To nato podamo funkciji `fmin` iz knjižnice SciPy, ki uporablja Nelder-Mead simpleksni algoritem za iskanje minimuma, začenši z začetno točko $(-5,5)$. Rešitev zapišemo:

```
1 def F( arg ):
2     x = arg[0]; y = arg[1]
3     return 100*(y-x**2)**2+(1-x)**2
4
5 Nfeval = 1
6 def callbackF(Xi):
7     global Nfeval
8     print('{0:4d}   {1: 3.6f}   {2: 3.6f}   {3: 3.6f}'.format(
9         Nfeval, Xi[0], Xi[1], F(Xi)))
10    Nfeval += 1
11
12 # uvozi funkcijo za simpleksno optimizacijo
13 from scipy.optimize import fmin
14
15 # definiraj funkcijo za izpis v posamezni iteraciji
16 print('{0:4s}   {1:9s}   {2:9s}   {3:9s}'.format(
17     'Iter', ' x', ' y', 'f(X)'))
18 # zaženi optimizacijo
19 xopt = fmin(func=F, x0=np.array([-5,5]), disp=1,
20            xtol=1e-6, ftol=1e-6, maxiter=1000, callback=callbackF)
```

```

21
22 print('\nOptimalne vrednosti:\n', xopt)

```

Ključni elementi klica funkcije `fmin` so:

- funkcija `callbackF`, ki vsako iteracijo izpiše številko iteracije, trenutne vrednosti parametrov in vrednost ciljne funkcije,
- nastavitve za natančnost (`xtol=1e-6`, `ftol=1e-6`) in maksimalno število iteracij (1000),
- spremljanje števila evalvacij funkcije s globalno spremenljivko `Nfeval`.

Rosenbrockova funkcija predstavlja težaven optimizacijski problem zaradi ozke in dolge doline, kar zahteva robustne optimizacijske metode. Simpleksni algoritem iterativno prilagaja velikost in obliko simpleksa (geometrične figure z $n + 1$ oglišči v n -dimenzionalnem prostoru), da bi našel minimum funkcije. Podatki o posamezni iteraciji in končne optimalne vrednosti se med in po klicu izpišejo:

```

Iter    x          y          f(X)
  1    -4.500000    5.375000    22156.812500
  2    -4.250000    5.937500    14729.125000
  3    -3.125000    6.468750    1103.954102
  4    -2.875000    7.031250    167.383789
  5    -2.875000    7.031250    167.383789
  6    -2.875000    7.031250    167.383789
  7    -2.875000    7.031250    167.383789
  8    -2.625000    7.031250    15.118164
...
      Current function value: 0.000000
      Iterations: 157
      Function evaluations: 285

Optimalne vrednosti:
[1.000000014 1.000000032]

```

Minimum Rosenbrockove funkcije je torej v točki $(x, y) = (1, 1)$, pri čemer je $F(1, 1) = 0$.

Vaja 8.4 Napišite funkcijo za izračun srednje kvadratne Evklidske razdalje med točkami (u_k, v_k) v zajeti sliki, ki so preslikane z modelom radialnih distorzij, in korespondenčnimi točkami (x_k, y_k) v metričnem prostoru, ki so preslikane s projektivno preslikavo:

$$E^2 = \frac{1}{N} \sum_{k=1}^N \|\mathcal{T}_{\text{radialna}}(\mathcal{T}_{\text{projektivna}}(x_k, y_k)) - (u_k, v_k)\|^2.$$

Funkcija naj kot prvi parameter vzame vrstični vektor `iPar`, ki predstavlja parametre 2D projektivne preslikave in modela radialnih distorzij kot $(a_{11}, a_{12}, t_x, a_{21}, a_{22}, t_y, p_x, p_y, u_c, u_v, K_1, K_2, \dots, K_n)$. Ostali parametri funkcije naj bodo N koordinat točk $(u_k, v_k) \rightarrow \text{iCoorU}$, `iCoorV`, in N koordinat točk $(x_k, y_k) \rightarrow \text{iCoorX}$, `iCoorY`:

```

1 def geomCalibErr( iPar, iCoorU, iCoorV, iCoorX, iCoorY ):
2     # Tu napišite python kodo funkcije
3     return oErr2

```

Funkcija naj izračuna napako med preslikanimi točkami E^2 , ki jo vrne kot skalarno spremenljivko `oErr2`. Začetni približek parametrov $(a_{11}, a_{12}, t_x, a_{21}, a_{22}, t_y)$ določite s pomočjo funkcije

za afino aproksimacijsko preslikavo `mapAffineApprox2D()` (Vaja 6).

Preizkusite delovanje funkcije tako, da naložite sliko `calibration-object.jpg` in ročno označite vsaj 8 oglišč na kalibracijskem objektu s pomočjo funkcije `matplotlib.pyplot.ginput()`. Za potrebe geometrijske kalibracije ustvarite referenčno mrežo točk, ki sovпада s položajem oglišč na idealni, ravninski sliki šahovnice. Referenčne točke naj ležijo v metričnem prostoru, pri tem pa predpostavite, da ima stranica kvadratnega polja šahovnice dolžino 20 mm. Naprimer, oglišče v levem zgornjem kotu šahovnice naj ima koordinati (20,20). Z uporabo simpleksne optimizacije določite optimalne parametre `iPar` tako, da bo napaka E^2 minimalna. Prikažite referenčne točke in točke oglišč na sliki pred in po preslikavi. ■

Pripravimo najprej funkcijo `geomCalibTrans` tako, da izvaja geometrijsko kalibracijsko preslikavo s kombiniranjem projektivne transformacije in radialne distorzije. V vektorju parametrov `iPar` naj najprej uporabi projektivno preslikavo s parametri `iPar[0:8]` za transformacijo iz originalnih koordinat (x, y) v vmesne koordinate, nato pa popravi radialne distorzije s parametri `iPar[8:]` (vključno s koordinatami optičnega centra (u_c, v_c) in koeficienti distorzije $\kappa_i, i = 1, 2, 3, \dots$). Funkcijo `geomCalibTrans` zapišemo:

```
1 def geomCalibTrans( iPar, iCoorX, iCoorY ):
2     """Funkcija za preslikavo točk (projektivna+radialne distorzije)"""
3     iParProj = iPar[0:8]
4     iParRad = iPar[8:]
5     # preslikava v prostor slike
6     iCoorUt, iCoorVt = transProjective2D( iParProj, iCoorX, iCoorY )
7     # korekcija radialnih distorzij
8     iCoorUt, iCoorVt = transRadial( iParRad[2:], iParRad[0], iParRad[1],
9                                     iCoorUt, iCoorVt )
10    # vrni preslikane točke
11    return iCoorUt, iCoorVt
```

Funkcija `geomCalibErr` nato izračuna srednjo kvadratno napako med dejanskimi koordinatami (u, v) in preslikanimi koordinatami (u_t, v_t) , kar omogoča optimizacijo kalibracijskih parametrov. Rešitev zapišemo kot:

```
1 def geomCalibErr( iPar, iCoorU, iCoorV, iCoorX, iCoorY ):
2     """Funkcija za izračun kalibracijske napake"""
3     # preslikaj točke z dano preslikavo
4     iCoorUt, iCoorVt = geomCalibTrans( iPar, iCoorX, iCoorY )
5     # izračun napake poravnave
6     oErr2 = np.mean( (iCoorU-iCoorUt)**2 + (iCoorV-iCoorVt)**2 )
7     # vrni vrednost napake
8     return oErr2
```

Ta funkcija se nato uporablja za natančno geometrijsko kalibracijo kamer, kjer je potrebno kompenzirati tako perspektivne deformacije kot tudi nelinearne optične popačitve. Za namen optimizacije je potrebno določiti pripadajoče pare točk, tj. točke na modelu kalibra in točke v vhodni sliki kalibra. V prvem primeru moram poznati dimenzije kalibra in polj šahovnice; izhodišče (0, 0) postavimo v gornji levi kot in v milimetrih zapišemo koordinate točk. V vhodni sliki interaktivno z miško naklikamo točke v istem zaporedju in jih shranimo:

```
1 # koordinate kalibra v metričnem prostoru v enoti mm
2 iCoorX = np.array([20,140,260,260,260,140,20,20])
3 iCoorY = np.array([20,20,20,100,180,180,100,100])
4 # naklikamo pripadajoče točke na kalibru
```

```

5 showImage( colorToGray(iCalImage), 'Slika kalibra' )
6 pts = plt.ginput(n=8, timeout=-1)
7 pts = np.array(pts)
8 # koordinate v prostoru slike
9 iCoorU = pts[:,0].flatten()
10 iCoorV = pts[:,1].flatten()

```

Koordinate ustrezno preoblikujemo in uporabo funkcijo za poravnavo pripadajočih parov točk z afino aproksimacijo `mapAffineApprox2D()` iz poglavja 7. S funkcijo `geomCalibErr()` določimo napako pred in po afini aproksimaciji:

```

1 # oblikujemo koordinate v matrike (Nx2)
2 ptsUV = np.vstack( (iCoorU, iCoorV) ).transpose()
3 ptsXY = np.vstack( (iCoorX, iCoorY) ).transpose()
4 # določi začetni približek parametrov
5 oMatA = mapAffineApprox2D( ptsUV, ptsXY )
6 # preslikava identiteta
7 Uc = iCalImageG.shape[1]/2
8 Vc = iCalImageG.shape[0]/2
9 iParIdent = np.array([1,0,0,0,1,0,0,0,Uc,Vc,0])
10 # preslikava z afinim približkom
11 Uc = iCalImageG.shape[1]/2
12 Vc = iCalImageG.shape[0]/2
13 iParAffine = np.array([oMatA[0,0],oMatA[0,1],oMatA[0,2],\
14                       oMatA[1,0],oMatA[1,1],oMatA[1,2],\
15                       0,0,Uc,Vc,0])
16 # določi napako z danimi parametri
17 oErr2ident = geomCalibErr( iParIdent, iCoorU, iCoorV, iCoorX, iCoorY )
18 oErr2affine = geomCalibErr( iParAffine, iCoorU, iCoorV, iCoorX, iCoorY )
19 print('Začetna napaka: ', oErr2ident)
20 print('Napaka po afini aproksimaciji: ', oErr2affine)

```

V ukazni vrsti dobimo rezultat:

```

Začetna napaka: 708764.720110
Napaka po afini aproksimaciji: 3010.379361

```

Napaka se po afini aproksimaciji bistveno zmanjša, a jo lahko nadalje zmanjšamo z dodajanjem primernih prostostnih stopenj v poravnavo, ki odražajo fizikalni model popačenja, tj. zaporedni perspektivna preslikava in radialna distozija. Za ta namen minimiziramo funkcijo `geomCalibErr()` s simpleksno optimizacijo, pri čemer kot začetni približek uporabimo parametre, ki smo jih pridobili z afino aproksimacijo:

```

1 # nacrtaj funkcijo za izpisovanje vrednosti parametrov
2 Nfeval = 1
3 def callbackF(Xi):
4     global Nfeval
5     np.set_printoptions(formatter={'float': '{: 0.3f}'.format})
6     formattedline = '%6.3f %6.3f %6.3f %6.3f %6.3f %6.3f \
7                     %8.3e %8.3e %8.3f %8.3f %8.3e' % ( tuple(Xi[:]) )
8     print('{0:4d} {1} {2: 3.6f}'.format(Nfeval, formattedline, F(Xi)))
9     Nfeval += 1
10
11 # uporabi simpleksno optimizacijo

```

```

12 Nfeval = 1
13 from scipy.optimize import fmin
14 F = lambda x: geomCalibErr( x, iCoorU, iCoorV, iCoorX, iCoorY )
15 print('{0:4s}   {1:20s}   {2:9s}'.format('Iter',
16     '[a11 a12 tx a21 a22 ty px py uc vc k]', 'f(X)'))
17 iParOpt = fmin(func=F, x0=iParAffine, maxiter=1000, disp=1, xtol=1e-6,
18     ftol=1e-6, callback=callbackF, maxfun=None )
19 formattedline1 = '%6.3f %6.3f %6.3f %6.3f %6.3f %6.3f \
20     %8.3e %8.3e %8.3f %8.3f %8.3e' % ( tuple(iParAffine[:]) )
21 formattedline2 = '%6.3f %6.3f %6.3f %6.3f %6.3f %6.3f \
22     %8.3e %8.3e %8.3f %8.3f %8.3e' % ( tuple(iParOpt[:]) )
23
24 oErr2opt = geomCalibErr( iParAffine, iCoorU, iCoorV, iCoorX, iCoorY )
25 print('Napaka po nelinearni optimizaciji: ', oErr2ident)
26 print('Začetni parametri (po afini aproks.):\\n',formattedline1)
27 print('Končni parametri (po nelin. opt.):\\n',formattedline2)

```

Rezultat se izpiše v ukazni vrstici:

```

---
Iter   [a11 a12 tx a21 a22 ty px py uc vc k]   f(X)
  1     4.816   0.631 -55.664 -0.936   4.562  202.175           0.000e+00
      ↳ 0.000e+00   785.000   578.000   0.000e+00   3010.379362
...
 10     4.835   0.636 -56.082 -0.943   4.597  203.694           1.796e-05
      ↳ -1.249e-04   790.896   582.341   3.756e-05   2949.440012
...
 20     4.795   0.626 -56.008 -0.933   4.544  199.215           5.966e-05
      ↳ -1.287e-04   831.258   546.378   1.459e-04   2577.553598
...
998     4.305  -0.418  67.167 -0.742   3.183  242.060           2.078e-04
      ↳ -1.364e-03  1441.124  1709.991   1.520e-02   2.866062
999     4.305  -0.419  67.224 -0.742   3.184  241.876           2.084e-04
      ↳ -1.364e-03  1446.320  1705.755   1.519e-02   2.865291
---
Napaka po nelinearni optimizaciji: 2.865291
Začetni parametri (po afini aproks.):
4.816   0.631 -55.664 -0.936   4.562  202.175           0.000e+00  0.000e+00
      ↳ 785.000   578.000   0.000e+00
Končni parametri (po nelin.opt.):
4.305  -0.419  67.224 -0.742   3.184  241.876           2.084e-04 -1.364e-03
      ↳ 1446.320  1705.755   1.519e-02

```

Napaka po nelinearni optimizaciji se drastično zniža iz 3010,379361 na 2,865291, kar kaže na dobro prilaganje modela preslikave za dani problem poravnave. Prikažimo še točke pred in po afini preslikavi in po simpleksni optimizaciji nelinearne perspektivne preslikave in radialnih distozij na naslednji način:

```

1 plt.imshow(iCalImageG, cmap = cm.Greys_r) # prikaži sliko v novem oknu
2 plt.suptitle('Slika kalibra s točkami za kalibracijo') # nastavi naslov slike
3 plt.xlabel('x'); plt.ylabel('y')
4 plt.plot(iCoorU,iCoorV,'om',markersize=5.0)
5 iCoorUt, iCoorVt = geomCalibTrans( iParIdent, iCoorX, iCoorY )
6 plt.plot(iCoorUt,iCoorVt,'or',markersize=5.0)
7 iCoorUt, iCoorVt = geomCalibTrans( iParAffine, iCoorX, iCoorY )
8 plt.plot(iCoorUt,iCoorVt,'ob',markersize=5.0)

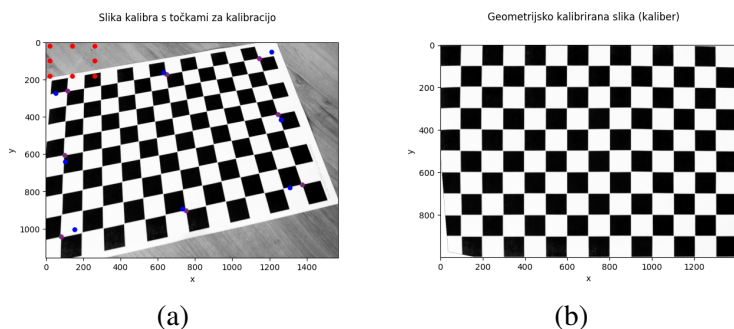
```

```

9 | iCoorUt, iCoorVt = geomCalibTrans( iParOpt, iCoorX, iCoorY )
10 | plt.plot(iCoorUt,iCoorVt,'+g',markersize=5.0)

```

Rezultat prikazuje slika 8.7, iz katere je razvidno dobro ujemanje točk po poravnavi z modelom perspektivne projekcije in radialnih distorzij.



Slika 8.7: (a) Slika kalibra z označenimi referenčnimi točkami (*vijolične točke*), referenčnih točk v metričnem prostoru kalibra (*rdeče točke*), ki smo jih nato z afino preslikavo preslikali v *modre točke* in nato še z zaporednima perspektivno preslikavo in radialnimi distorzijami po minimizaciji napake (*modre točke nad oz. tik ob vijoličnih točkah*). (b) Področje šahovnice preslikano v metrični prostor, pri čemer je velikost posameznega slikovnega elementa 0,2 mm.

8.5 Naloge in vprašanja

Poročilo naloge pripravite v obliki Python skripte, ki naj izvede zahtevane izračune oz. klice funkcij in izriše pripadajoče slike oz. grafe. Vaša dolžnost je, da v skripti uvozite ustrezne Python knjižnice in dodate morebitne podporne skripte. Skripta se mora izvesti in poustvariti rezultate, kot se zahtevajo v nalogah. Na vprašanja odgovorite v obliki komentarja, naprimer *# Naloga X.1: To je moj odgovor..*

Naloga 8.1 Napišite funkcijo, ki za dane optimalne parametre geometrijske kalibracije *iPar* normalizira zajeto sliko *iImage* v metrični prostor:

```

1 | def geomCalibImage( iPar, iImage, iCoorX, iCoorY ):
2 |     # Tu napišite python kodo funkcije
3 |     return oImage

```

kjer spremenljivki *iCoorX* in *iCoorY* predstavljata koordinate točk (x,y) v **metričnem prostoru**, ki jih je potrebno preslikati v prostor slike in določiti pripadajoče sivinske vrednosti. Funkcija vrne kalibrirano sliko *oImage*, katere dimenzije so podane glede na dimenzije vzorčnih točk (x,y) .

Preverite delovanje funkcije tako, da vzorčne točke (x,y) preslikate z optimalnimi parametri projektivne preslikave in modela radialnih distorzij. Predpostavite, da ima stranica kvadratnega polja kalibracijskega objekta dolžino 20 mm in ustvarite točke (x,y) tako, da bo na izhodni kalibrirani sliki *oImage* velikost slikovnih elementov enaka $1/5$ mm. Izvedite geometrijsko kalibracijo slike *calibration-object.jpg* tako, da s funkcijo za interpolacijo prvega reda *scipy.interpolate.interpn()* v preslikanih točkah poiščete nove sivinske vrednosti. Prikažite preslikano sliko, v kateri morajo biti vzorci v kalibru poravnani z osmi slike.

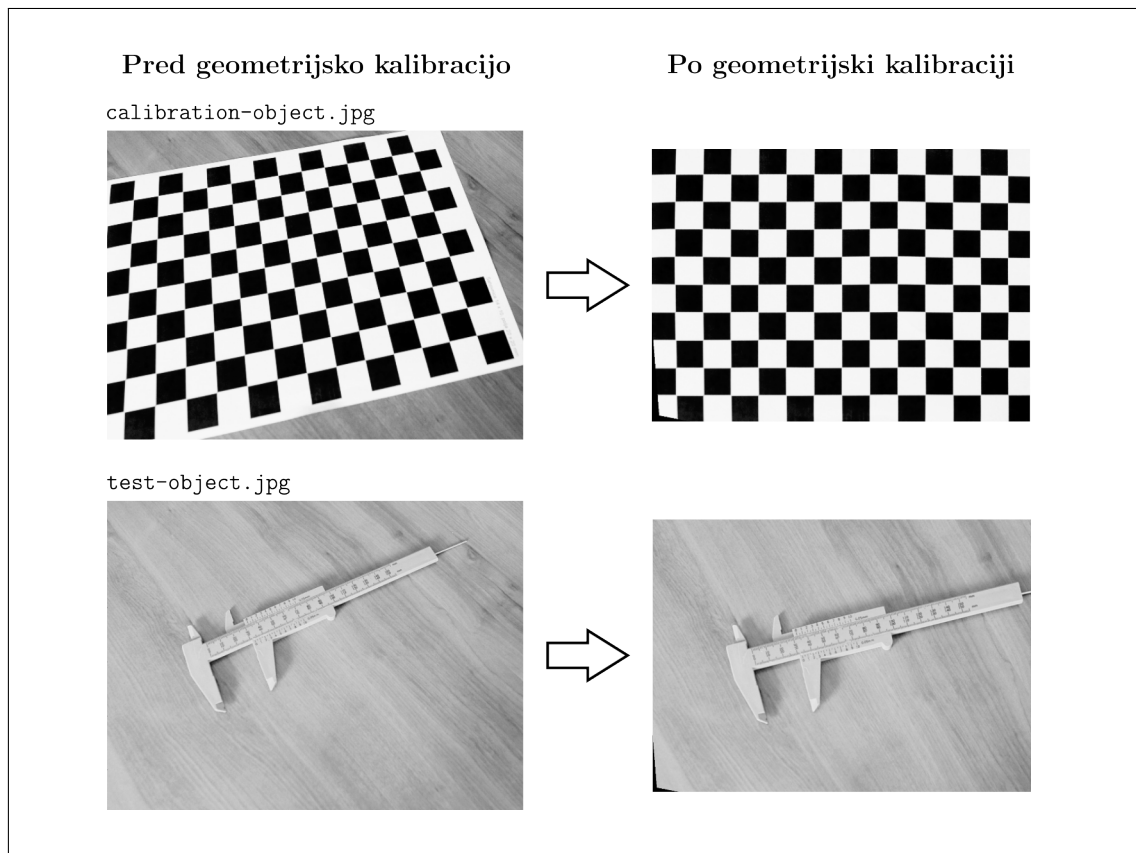
Naloga 8.2 Barvno sliko objekta *test-object.jpg*, ki je bila zajeta z istim slikovnim sistemom kot slika kalibra, kalibrirajte z referenčno geometrijsko kalibracijo tako, da bo velikost slikovnega

elementa enaka $1/3$ mm. Prikažite kalibrirano barvno sliko in nato na merilu označite daljico med kljunoma kljunastega merila ter izračunajte njeno dolžino v mm. Preverite ali dolžina ustreza dolžini, ki je označena na kljunastem merilu.

Naloga 8.3 Geometrijsko kalibracijo ponovite $3 \times$ tako, da najprej $3 \times$ ročno označite oglišča na sliki s kalibracijskim objektom `calibration-object.jpg`. Referenčno geometrijsko kalibracijo nato določite glede na povprečne koordinate teh oglišč. Na podlagi referenčne geometrijske kalibracije ocenite točnost geometrijske kalibracije v mm z ročno označenimi oglišči. Od česa zavisi točnost geometrijske kalibracije?

Naloga 8.4 Razvijte avtomatski postopek za geometrijsko kalibracijo slikovnega sistema. S pomočjo Harrisovega detektorja oglišč iz knjižnice OpenCV (Vaja 6) poiščite oglišča na sliki kalibra `calibration-object.jpg`. Nato pa s funkcijo `cv2.cornerSubPix()` še dodatno izpopolnite položaj oglišč. Ustvarite idealno mrežo referenčnih oglišč in s postopkom ICP (Vaja 5) ter afino preslikavo določite korespondence z zaznanimi oglišči na sliki kalibra. Določite natančnost avtomatske v primerjavi z referenčno geometrijsko kalibracijo v mm. Poizkusite izboljšati natančnost kalibracije še tako, da avtomatsko glede na lastne pogoje zavržete korespondence, ki slabšajo kalibracijo.

Rezultati nekaterih nalog:





9. Določanje poze objektov v prostoru

Prileganje 3D modelov ali slik objekta zanimanja na 2D slike omogoča določanje položaja teh objektov v 3D prostoru. Postopek prileganja 3D modela na 2D sliko imenujemo **3D-2D poravnava**. 3D-2D poravnava je proces iskanja optimalne geometrijske preslikave $\mathcal{T} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$, ki 3D model oz. sliko V nekega objekta preslika v tako lego, ki je *skladna* s projekcijo istega objekta na zajeti 2D sliko P . Glavni izziv pri poravnavi 3D in 2D slik je prostorska neskladnost slikovne informacije (3D vs. 2D), ki jo lahko odpravimo na dva načina: 1) s **projekcijo 3D informacije v 2D slikovni prostor** ali 2) z rekonstrukcijo 3D slike iz različnih 2D projekcij. Pri vaji boste načrtali postopek poravnave s projekcijo 3D slike v 2D ($\mathcal{P} : \mathbb{R}^3 \rightarrow \mathbb{R}^2$) in maksimizacijo mere podobnosti MP med projekcijo $\mathcal{P}(V)$ in 2D sliko P . Gradniki postopka so prikazani na sliki 9.1 *spodaj*.

9.1 Projekcija iz 3D v 2D

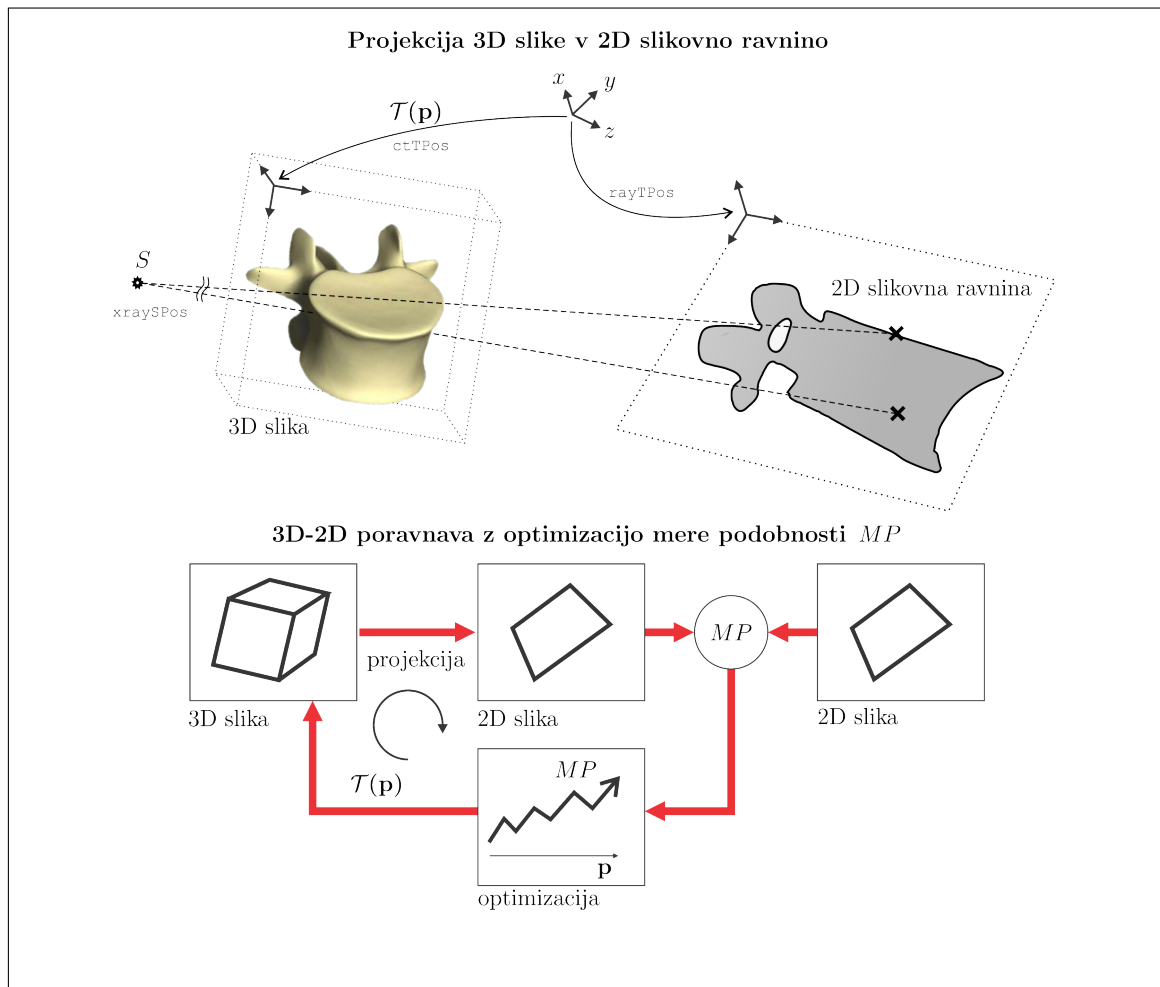
Način projekcije zavisi od oblike 3D informacije. Pri vaji bo dana 3D CT¹ slika, v kateri vsak slikovni element predstavlja atenuacijski koeficient $\mu(\mathbf{x})$ zajete 3D scene. Za potrebe poravnave 3D in 2D slik boste s pomočjo 3D CT slike simulirali 2D rentgenske slike, pri katerih se projekcija iz 3D prostora v 2D sliko izraža kot integral atenuacijskega koeficienta vzdolž premice od izvora rentgenskih žarkov S do 2D slikovne ravnine $P = \int_l \mu(l) dl$. Tovrstno projekcijo 3D CT slike imenujemo digitalno rekonstruirani rentgenski posnetek (DRR²). Projekcija in pripadajoče količine so prikazane na sliki 9.1 *zgoraj*.

9.2 Mera podobnosti

Prostorsko ujemanje med projekcijo $\mathcal{P}(V)$ in 2D sliko P ovrednotimo z mero podobnosti MP ; to je skalarna funkcija, ki zavzame maksimum, ko se položaji korespondenčnih struktur v $\mathcal{P}(V)$ in P medsebojno prekrivajo. Mero podobnosti je potrebno smiselno izbrati, tako da je čim manj občutljiva na motilna slikovna neskladja in čim bolj občutljiva na dejanska geometrijska neskladja

¹CT – ang. Computed Tomography

²DRR – ang. Digitally Reconstructed Radiograph



Slika 9.1: Zagotavljanje dimenzijskega ujemanja 3D in 2D slik s projekcijo 3D slike na 2D ravnino (zgoraj) in gradniki avtomatskega algoritma 3D-2D poravnave (spodaj).

med slikama. Za poravnavo slik se pogosto uporablja **medsebojna informacija** MI^3 :

$$MI(I, J) = H(I) + H(J) - H(I, J), \quad (9.1)$$

kjer sta $H(I)$ in $H(J)$ entropija referenčne $I(x, y)$ in lebdeče $J(x, y)$ slike, $H(I, J)$ pa njuna skupna entropija:

$$\begin{aligned} H(I) &= - \sum_{s_I=0}^{L-1} p_I(s_I) \log p_I(s_I), \\ H(J) &= - \sum_{s_J=0}^{L-1} p_J(s_J) \log p_J(s_J), \\ H(I, J) &= - \sum_{s_I=0}^{L-1} \sum_{s_J=0}^{L-1} p_{IJ}(s_I, s_J) \log p_{IJ}(s_I, s_J), \end{aligned} \quad (9.2)$$

kjer spremenljivki s_I in s_J označujeta soležne, diskretne sivinske vrednosti referenčne slike $I(x, y)$ in lebdeče slike $J(x, y)$, L pa število diskretnih vrednosti. Verjetnostni porazdelitvi $p_I(s_I)$ in $p_J(s_J)$ ter

³MI – ang. Mutual Information

skupno porazdelitev $p_{IJ}(s_I, s_J)$ dobimo iz pripadajočih normaliziranih histogramov $h_I(s_I)$, $h_J(s_J)$ ter $h_{IJ}(s_I, s_J)$:

$$p_I(s_I) = \frac{h_I(s_I)}{N \cdot M}, \quad p_J(s_J) = \frac{h_J(s_J)}{N \cdot M}, \quad p_{IJ}(s_I, s_J) = \frac{h_{IJ}(s_I, s_J)}{N \cdot M}, \quad (9.3)$$

kjer N in M predstavljata število elementov vzdolž x in y osi slike.

9.3 Optimizacijski postopek

Optimizacija je ključen del številnih znanstvenih in inženirskih disciplin, saj omogoča iskanje optimalnih rešitev za različne probleme. Optimizacijske metode lahko razdelimo v naslednje kategorije:

- **Gradientne metode:** uporabljajo odvode za iskanje ekstremov funkcij.
- **Evolucijske in heuristične metode:** temeljijo na simuliranem naravnem izboru.
- **Kombinatorične metode:** iščejo optimalne rešitve v diskretnih prostorih.

V Pythonu imamo na voljo več knjižnic za implementacijo optimizacijskih algoritmov:

- `scipy.optimize`: nudi številne metode, kot so gradienti spust (ang. *gradient descent*), simpleksna metoda *Nelder-Mead*, *BFGS* in druge.
- `cvxpy`: orodje za konveksno optimizacijo.
- `pulp`: rešuje linearne optimizacijske probleme.
- `DEAP`: uporablja evolucijske algoritme.

9.3.1 Primer: optimizacija nelinearne funkcije

Spodaj je prikazan primer uporabe metode *BFGS* iz knjižnice `scipy.optimize` za minimizacijo nelinearne funkcije:

```
1 import numpy as np
2 from scipy.optimize import minimize
3
4 # Definicija funkcije
5 def objective_function(x):
6     return x**4 - 3*x**3 + 2
7
8 # Začetna točka
9 x0 = np.array([2.0])
10
11 # Optimizacija z metodo BFGS
12 result = minimize(objective_function, x0, method='BFGS')
13
14 # Izpis rezultata
15 print("Optimalna vrednost x:", result.x)
16 print("Minimalna vrednost funkcije:", result.fun)
```

V tem primeru funkcija $f(x) = x^4 - 3x^3 + 2$ vsebuje lokalne ekstreme. Metoda *BFGS* iterativno prilagaja vrednosti x in poišče minimum funkcije.

9.3.2 Optimizacija mere podobnosti za 3D togo preslikavo

Tekom poravnave 3D in 2D slike nam izbrana optimizacijska metoda iterativno spreminja parametre $\mathbf{p}$ geometrijske preslikave $\mathcal{T} = \mathcal{T}(\mathbf{p})$ tako, da maksimizira mero podobnosti:

$$\mathbf{p}^* = \operatorname{argmax}_{\mathbf{p}} MP\left(\mathcal{P}(V[\mathcal{T}(\mathbf{p})]), P\right), \quad (9.4)$$

kjer so $\mathbf{p}^*$ optimalni parametri preslikave $\mathcal{T}(\mathbf{p})$. Za poravnavo 3D in 2D slik vretenč boste uporabili togo preslikavo, ki je v 3D definirana s šestimi parametri $\mathbf{p} = [t_x, t_y, t_z, \alpha, \beta, \gamma]^T$. Postopek poravnave 3D in 2D slik je prikazan na gornji sliki.

9.4 Vaje z rešitvami

Gradivo za vajo v datoteki `vretenca.npz` vsebuje dve 3D CT sliki in eno 2D rentgensko sliko. Datoteko naložite z ukazom `numpy.load()`, z branjem lastnosti `files` pa lahko preverite imena vsebovanih spremenljivk. Vse spremenljivke so dane v obliki numpy polja. V spremenljivkah `ctVol` in `ct2Vol` ter `xrayImg` so dane pripadajoče 3D in 2D matrike sivinskih vrednosti, v spremenljivkah `ctTPos`, `ct2TPos` in `xrayTPos` geometrijske preslikave koordinatnih sistemov 3D in 2D slik iz referenčnega koordinatnega sistema v prvi slikovni element slike z indeksom `[0, 0, 0]` oziroma `[0, 0]`, v spremenljivki `xraySPos` pa položaj izvora rentgenskih žarkov v referenčnem koordinatnem sistemu. Korak vzorčenja 3D in 2D slik je 1 milimeter. Geometrijske razmere in količine so ponazorjene v gornji sliki.

Koda vaje je dostopna na [Git repozitoriju](#).

Uvozimo Python knjižnice in pripravimo pomožne funkcije:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import time
4 import rvlib
5
6 from scipy.optimize import fmin, minimize
7 from scipy.interpolate import interpn
8 from mpl_toolkits.mplot3d import Axes3D
9
10 def preprocess_ct(ct):
11     ct_temp = dict(ct)
12     ct_temp['img'] = (ct_temp['img'] - np.median(ct_temp['img'])).\
13         astype('float')
14     return ct_temp
15
16 def preprocess_Xray(Xray):
17     Xray_temp = dict(Xray)
18     Xray_temp['img'] = rvlib.windowImage(Xray_temp['img'], 60.0, 120.0)
19     Xray_temp['img'] = Xray_temp['img'].astype('float') / \
20         Xray_temp['img'].max() * 255.0
21     return Xray_temp
```

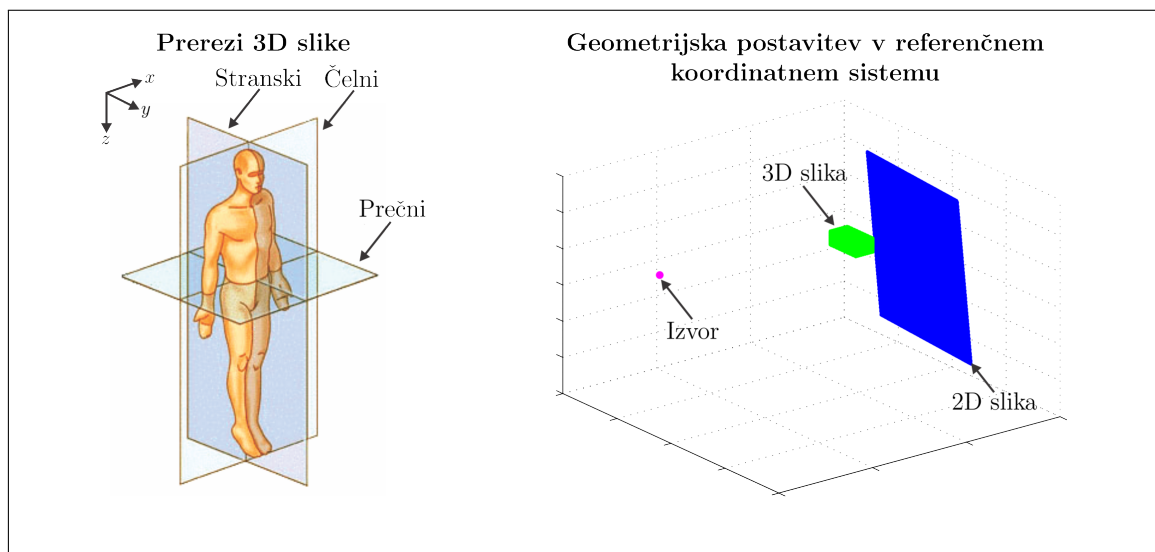
Funkciji `preprocess_ct()` in `preprocess_Xray()` izvajata predhodno obdelavo medicinskih slikovnih podatkov za CT in rentgenske posnetke. Funkcija `preprocess_ct()` najprej ustvari kopijo vhodnega slovarja CT slike, nato pa normalizira slikovne vrednosti tako, da odšteje mediano vrednost slike (`np.median(ct_temp['img'])`), kar centrira podatke okoli ničle, in rezultat pretvori v tip `float`. Funkcija `preprocess_Xray()` prav tako ustvari kopijo vhodnega slovarja rentgenskega posnetka, nato pa uporabi oknjenje slike (`rvlib.windowImage()`) z oknom 60-120 HU za izboljšavo kontrasta, nakar slikovne vrednosti normalizira v območje 0-255 z deljenjem z maksimalno vrednostjo in množenjem s 255,0, kar omogoči primerjavo z drugimi slikami. Obe funkciji ohranjata originalne metapodatke v slovarju, medtem ko modificirata le slikovne podatke, kar je ključno za nadaljnjo analizo v medicinskih aplikacijah.

Vaja 9.1 Za potrebe nadaljnjih nalog za vsako od danih slik ustvarite spremenljivko tipa dict, npr. `ct`, `ct2` in `Xray`, in pod ustreznim ključem vnesite sivinske vrednosti slike in pripadajočo preslikavo v referenčnem koordinatnem sistemu (`ctTPos`, `ct2TPos` in `xrayTPos`), za 2D rentgensko sliko pa še položaj izvora rentgenskih žarkov (`xraySPos`). ■

Naložimo datoteko `vretenca.npz` in oblikujemo slovarje tako, da pod ključem `'img'` shranimo polje sivinski vrednosti slike, pod `'TPos'` koordinatni sistem (preslikavo iz svetovnega koordinatnega sistema v prvo točko slike), in pod `'SPos'` koordinate izvora rentgenskih žarkov:

```
1 v = np.load('vretenca.npz')
2 # oblikuj v dict
3 Xray = {'img':v['xrayImg'], 'TPos':v['xrayTPos'], 'SPos':v['xraySPos']}
4 ct = {'img':v['ctVol'], 'TPos':v['ctTPos']}
5 ct2 = {'img':v['ct2Vol'], 'TPos':v['ct2TPos']}
```

Vaja 9.2 Ustvarite vzorčni mreži 2D rentgenske in 3D CT slike s pomočjo ukaza `numpy.meshgrid()`. Točke na vzorčnih mrežah preslikajte v referenčni koordinatni sistem s pripadajočimi 3D togimi preslikavami `Xray.TPos` in `ct.TPos`. Preslikane točke 3D in 2D vzorčnih mrež in položaj izvora rentgenskih žarkov `Xray.SPos` prikažite s pomočjo ukaza `Axes3D.scatter()`. Preverite pravilnost dobljene geometrijske postavitve izvora *S*, 3D in 2D slike v referenčnem koordinatnem sistemu s pomočjo slike 9.2. ■



Slika 9.2: Geometrijske postavitev izvora *S*, 3D in 2D slike v referenčnem koordinatnem sistemu.

Izvedemo prostorsko vizualizacijo transformacije koordinat med 3D CT sliko (`ct`) in 2D rentgenskim posnetkom (`Xray`) tako, da najprej ustvarimo mrežo 3D koordinat (`g3`) iz dimenzij CT slike z uporabo `np.meshgrid`, ki jo pretvorimo v homogeni koordinatni sistem z dodajanjem enice v zadnji stolpec. Analogno ustvarimo 2D mrežo koordinat za rentgenski posnetek (`g2`), ki jo razširimo v homogeni prostor z ničlo v tretji in enico v četrti komponenti. Nato izvedemo transformacijo koordinat z uporabo transformacijskih matrik `ct['TPos']` in `Xray['TPos']`, ki preslikajo točke v skupni prostorski referenčni sistem. Prikažemo še vir rentgenskega žarka (`xsp`) z vijoličnim krogcem, preslikane 3D točke CT (zeleno) in preslikane 2D točke rentgenskega posnetka (modro), pri čemer se zaradi preglednosti uporablja podvzorčenje s faktorji `s3` in `s2`. Rešitev

zapišemo:

```

1 s3z,s3y,s3x = ct['img'].shape
2 g3x,g3y,g3z = np.meshgrid(range(s3x),range(s3y),range(s3z),indexing='xy')
3 # ustvari Nx4 matriko, zadnja koordinata homogena
4 g3 = np.vstack((g3x.flatten(),g3y.flatten(),g3z.flatten())).transpose()
5 g3 = np.hstack((g3, np.ones((g3.shape[0],1))))
6
7 s2y,s2x = Xray['img'].shape
8 g2x,g2y = np.meshgrid(range(s2x),range(s2y),indexing='xy')
9 # ustvari Mx4 matriko, zadnja koordinata homogena
10 g2 = np.vstack((g2x.flatten(),g2y.flatten())).transpose()
11 g2 = np.hstack((g2, np.zeros((g2.shape[0],1)), np.ones((g2.shape[0],1))))
12
13 g3p = np.dot(g3, ct['TPos'].transpose())
14 g2p = np.dot(g2, Xray['TPos'].transpose())
15 xsp = Xray['SPos'].flatten()
16
17 s2 = 50
18 s3 = 30
19
20 plt.close('all')
21 fig = plt.figure()
22 ax = fig.add_subplot(111,projection='3d')
23 ax.scatter(xsp[0],xsp[1],xsp[2], c='m', marker='o')
24 ax.scatter(g2p[:,s2,0],g2p[:,s2,1],g2p[:,s2,2], c='b', marker='.',linewidths=0)
25 ax.scatter(g3p[:,s3,0],g3p[:,s3,1],g3p[:,s3,2], c='g', marker='.',linewidths=0)
26 plt.show()

```

3D prikaz omogoča analizo relativne geometrije med obema modalitetama, kar je ključno za primerjavo ali poravnavo slik. Rezultat je prikazan na sliki 9.2 *desno*.

Vaja 9.3 Napišite funkcijo, ki preslika poljubno točko $iPos$ v 3D prostoru na 2D slikovno ravnino glede na izvor rentgenskih žarkov S (xraySPos):

```

1 def mapPointToPlane( iPos, Xray ):
2     # Tu napišite python kodo funkcije
3     return oPos

```

kjer je $iPos$ matrika $N \times 3$, $Xray$ pa spremenljivka s podatki o 2D rentgenski sliki. Funkcija vrne točke v matriki $oPos$ v obliki matrike $N \times 3$. V ravnino preslikane točke izračunate kot presečišča 2D slikovne ravnine s premicami, ki izvirajo v izvoru rentgenskih žarkov in gredo skozi točke $iPos$. Preverite delovanje funkcije tako, da preslikate koordinate oglišč 3D slike na 2D slikovno ravnino in točke vrišete v geometrijsko postavitev iz prejšnje naloge. ■

Funkcija `mapPointToPlane()` izvaja projekcijo 3D točk na ravnino detektorja rentgenskega sistema z uporabo geometrije cone-beam projekcije. Najprej normalizira vhodne koordinate in določi referenčno točko ($p0$) ter normalo (n) na ravnino detektorja z uporabo transformacijske matrike `Xray['TPos']`. Projekcijska geometrija temelji na konceptu centralne projekcije, kjer se vsaka 3D točka ($iPos$) preslika na detektor vzdolž žarka, ki poteka od izvora žarkov (`Xray['SPos']`) skozi obravnavano točko. Ključni korak je izračun skalirnega faktorja α , ki določa presečišče tega žarka z ravnino detektorja. Rezultat so 3D koordinate točk na detektorju, ohranjene v prostorskem koordinatnem sistemu. Rešitev zapišemo:

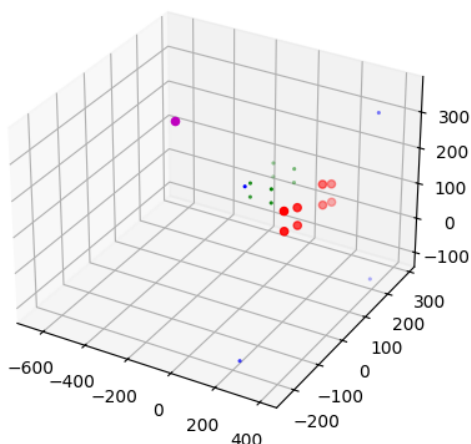

```

1 def mapPointToPlane(iPos, Xray):
2     # točke so v obliki Nx3/4
3     iPos = np.asarray(iPos)
4     if iPos.ndim == 1:
5         iPos = np.reshape(iPos, (1,iPos.size))
6     iPos = iPos[:, :3]
7     # določi izhodišče na detektorji
8     p0 = np.dot(Xray['TPos'], np.array([0, 0, 0, 1]))
9     # določi normalo na ravnino
10    ex = np.dot(Xray['TPos'], np.array([1, 0, 0, 1]))
11    ey = np.dot(Xray['TPos'], np.array([0, 1, 0, 1]))
12    n = np.cross(ex[:-1]-p0[:-1], ey[:-1]-p0[:-1])
13    n = n / np.sqrt(np.sum(n**2.0))
14
15    # skrajšaj vse vektorje na dimenzije 1x3
16    p0 = p0[:3]; n = n[:3]
17    s = Xray['SPos'].reshape((1,3))
18
19    # koeficient za skaliranje smernega vektorja
20    alpha = np.dot(p0 - s, n) / np.dot(iPos - s, n)
21
22    # določi položaj točk na detektorju
23    oPos = s + alpha.reshape((iPos.shape[0],1)) * (iPos - s)
24
25    return oPos
26
27 # ustvari mrežo točk oglišč 3D slike
28 sz,sy,sx = ct['img'].shape
29 g3x, g3y, g3z = np.meshgrid((0,sx-1),(0,sy-1),(0,sz-1),indexing='xy')
30 g3 = np.vstack((g3x.flatten(),g3y.flatten(),g3z.flatten())).transpose()
31 g3 = np.hstack((g3, np.ones((g3.shape[0],1))))
32 g3p = np.dot(g3, ct['TPos'].transpose())
33
34 # ustvari mrežo točk oglišč 2D slike
35 s2y,s2x = Xray['img'].shape
36 g2x,g2y = np.meshgrid((0,s2x-1),(0,s2y-1),indexing='xy')
37 # ustvari Mx4 matriko, zadnja koordinata homogena
38 g2 = np.vstack((g2x.flatten(),g2y.flatten())).transpose()
39 g2 = np.hstack((g2, np.zeros((g2.shape[0],1)), np.ones((g2.shape[0],1))))
40 g2p = np.dot(g2, Xray['TPos'].transpose())
41
42 # koordinata izvora zarkov
43 xsp = Xray['SPos'].flatten()
44
45 # točke 3D oglišč na ravnini detektorja
46 g2proj = mapPointToPlane(g3p[:, :3], Xray)
47
48 # nariši geometrijske razmere
49 s = 1
50 plt.close('all')
51 fig = plt.figure()
52 ax = fig.add_subplot(111,projection='3d')
53 ax.scatter(xsp[0],xsp[1],xsp[2], c='m', marker='o')
54 ax.scatter(g2p[:, :s,0],g2p[:, :s,1],g2p[:, :s,2], c='b', marker='.',linewidths=0)
55 ax.scatter(g3p[:, :s,0],g3p[:, :s,1],g3p[:, :s,2], c='g', marker='.',linewidths=0)
56 ax.scatter(g2proj[:, :0],g2proj[:, :1],g2proj[:, :2], c='r', marker='o')

```

```
57 | plt.show()
```

Testna koda vizualizira geometrijske razmere med CT volumom in rentgenskim sistemom. Najprej generira mrežo 3D točk (oglišča CT volumna) in 2D točk (pikslov detektorja), ki jih transformira v skupni koordinatni sistem. Nato uporabi `mapPointToPlane()` za projekcijo 3D točk na ravnino detektorja. Projicirane točke morajo ležati v ravnini. Prikaz na sliki 9.3 omogoča preverjanje pravilnosti projekcijske geometrije in transformacijskih matrik.



Slika 9.3: Vizualizacija prikazuje: izvor žarkov (vijolična), detektor (modre točke), CT volumen (zeleno) in projicirane točke (rdeče).

Vaja 9.4 Napišite funkcijo za stožčasto projekcijo 3D CT slike v 2D slikovno ravnino, ki simulira delovanje rentgena (DRR; ang. *Digitally Reconstructed Radiograph*):

```
1 def project3DTo2D( ct, Xray, iStep ):
2     # Tu napišite python kodo funkcije
3     return oDRR, oMask
```

kjer sta `ct` in `Xray` spremenljivki s podatki o 3D CT in 2D rentgenski sliki, `iStep` pa je korak vzorčenja v milimetrih vzdolž vsake premice od izvora rentgenskih žarkov do presečišča z 2D slikovno ravnino. Funkcija vrne 2D DRR sliko v spremenljivki `oDRR` in 2D masko DRR slike v spremenljivki `oMask`, ki imata dimenzije 446×446 (enako kot `xrayImg`). 2D maska DRR slike označuje tiste točke 2D slikovne ravnine, za katere premica do izvora rentgenskih žarkov seče 3D CT sliko. Preizkusite delovanje funkcije s pomočjo danih `ct` in `Xray` slik in ustvarite DRR oz. projekcijo 3D CT slike. Preizkusite vpliv dolžine koraka `iStep` na projicirano sliko. ■

Funkcija `project3DTo2D()` izvaja stožčasto projekcijo 3D CT slike v 2D ravnino rentgenskega detektorja. Postopek se začne z generiranjem mreže 3D točk, ki predstavljajo oglišča CT volumna, in njihovo transformacijo v skupni koordinatni sistem z uporabo matrike `ct['TPos']`. Te točke se nato projicirajo na ravnino detektorja s funkcijo `mapPointToPlane()`, ki uporablja geometrijo centralne projekcije z izvorom žarkov `Xray['SPos']`. Določi se najmanjši omejujoči pravokotnik projiciranih točk na detektorju, kar omogoča optimizacijo računskega območja. Nato se generira mreža vzorčnih točk vzdolž žarkov, ki potekajo od izvora skozi vsak piksel detektorja, pri čemer se uporabi korak vzorčenja `iStep` za nadzor gostote vzorčenja.

Vzorčne točke v 3D prostoru se nato transformirajo nazaj v koordinatni sistem CT slike z inverzno matriko `ct['TPos']`, kjer se z interpolacijsko funkcijo (`interp()`) določijo ustrezne sivinske vrednosti iz CT volumna. Za točke znotraj CT volumna se izvede trilinearna interpolacija, medtem ko se točke zunaj obravnavajo kot ozadje (vrednost 0). Končna 2D projekcija (`oDRR`) nastane z integracijo (povprečenjem) interpoliranih vrednosti vzdolž vsakega žarka, maska (`oMask`) pa označuje območje, kjer je bila projekcija izvedena. Rezultat je simulacija rentgenskega posnetka, ki upošteva geometrijske lastnosti sistema in omogoča primerjavo z realnimi rentgenskimi posnetki.

Rešitev zapišemo z uporabo matričnih operacij preslikave koordinat, kar omogoča učinkovito izvedbo:

```

1 def project3DTo2D(ct, Xray, iStep):
2     '''
3     Naloga 4: Funkcija za stožčasto projekcijo 3D slike v 2D ravnino
4
5     Parameters
6     -----
7     ct : dict
8         Podatki o 3D ct sliki in sivinske vrednosti pod ključi TPos in img
9     Xray : dict
10        Podatki o 2D Xray sliki in sivinske vrednosti pod ključi TPos, SPos in
11        img
12    iStep : float
13        Korak vzorčenja v mm vzdolž Xray žarkov od izvora do 2D detektorja
14
15    Returns
16    -----
17    oDRR : numpy.ndarray
18        Izhodna iz 3D v 2D prostor preslikana slika, ki simulira rentgen
19    oMask : numpy.ndarray
20        Maska področja kamor se preslika 3D slika (1-ospredje, 0-ozadje)
21    '''
22    # ustvari mrežo točk oglišč 3D slike
23    s3z, s3y, s3x = ct['img'].shape
24    g3x, g3y, g3z = np.meshgrid((0,s3x-1),(0,s3y-1),(0,s3z-1),indexing='xy')
25    g3 = np.vstack((g3x.flatten(),g3y.flatten(),g3z.flatten())).transpose()
26    g3 = np.hstack((g3, np.ones((g3.shape[0],1))))
27    g3p = np.dot(g3, ct['TPos'].transpose())
28
29    # točke 3D oglišč na ravnini detektorja
30    g2proj = mapPointToPlane(g3p[:, :3], Xray)
31
32    # preslikaj v 2D ravnino
33    g2plane = np.dot(addHomogCoord(g2proj), \
34                     np.linalg.inv(Xray['TPos']).transpose())
35
36    # poišči najmanjši očrtan pravokotnik
37    xmin = np.floor(np.min(g2plane[:,0]))
38    xmax = np.ceil(np.max(g2plane[:,0]))
39    ymin = np.floor(np.min(g2plane[:,1]))
40    ymax = np.ceil(np.max(g2plane[:,1]))
41
42    # preveri ali so točke znotraj 2D slike
43    s2y, s2x = Xray['img'].shape
44    xmin = np.max((0,xmin)); xmin = int(np.min((s2x,xmin)))
45    xmax = np.max((0,xmax)); xmax = int(np.min((s2x,xmax)))

```

```

45     ymin = np.max((0,ymin)); ymin = int(np.min((s2y,ymin)))
46     ymax = np.max((0,ymax)); ymax = int(np.min((s2y,ymax)))
47
48     # definiraj mrežo točk v 2D ravnini
49     g2x, g2y = np.meshgrid(range(xmin,xmax), range(ymin,ymax), indexing='xy')
50     rectShape = g2x.shape
51
52     # ustvari Mx4 matriko, zadnja koordinata homogena
53     g2 = np.vstack((g2x.flatten(),g2y.flatten())).transpose()
54     g2 = np.hstack((g2, np.zeros((g2.shape[0],1)), np.ones((g2.shape[0],1))))
55     g2 = np.dot(g2, Xray['TPos'].transpose())
56
57     # preberi pozicijo izvora žarkov
58     xsp = Xray['SPos'].flatten().reshape((1,3))
59
60     # določi minimalno in maksimalno razdaljo za vzorčenje
61     d = np.sqrt(np.sum((g3p[:, :3] - xsp)**2.0, axis=1))
62     dmin = np.min(d); dmax = np.max(d)
63
64     # določi vzorčne točke vzdolž žarkov
65     ds = np.arange(dmin,dmax,iStep)
66     ds = np.reshape(ds, (1,ds.size))
67
68     # definiraj vzorčne točke v 3d prostoru
69     vs = g2[:, :3] - xsp
70     vs = vs / np.sqrt(np.sum(vs**2.0, axis=1)).reshape((vs.shape[0],1))
71
72     Nx1 = (vs.shape[0],1)
73     px = xsp[0,0] + vs[:,0].reshape(Nx1) * ds
74     py = xsp[0,1] + vs[:,1].reshape(Nx1) * ds
75     pz = xsp[0,2] + vs[:,2].reshape(Nx1) * ds
76
77     # preslikava koordinat v prostor 3D slike
78     Tmat = np.linalg.inv(ct['TPos'])
79     pxn = Tmat[0,0]*px + Tmat[0,1]*py + Tmat[0,2]*pz + Tmat[0,3]
80     pyn = Tmat[1,0]*px + Tmat[1,1]*py + Tmat[1,2]*pz + Tmat[1,3]
81     pzn = Tmat[2,0]*px + Tmat[2,1]*py + Tmat[2,2]*pz + Tmat[2,3]
82
83     # preveri katere koordinate so znotraj 3D slike
84     idx = np.where((pxn>=0) & (pxn<s3x) & \
85                   (pyn>=0) & (pyn<s3y) & \
86                   (pzn>=0) & (pzn<s3z))
87
88     oRayInterp = np.zeros_like(pxn)
89     pxn = pxn[idx[0],idx[1]]
90     pyn = pyn[idx[0],idx[1]]
91     pzn = pzn[idx[0],idx[1]]
92
93     # izvedi trilinearno interpolacijo
94     oRayInterp_i = interpn((np.arange(s3z),np.arange(s3y),np.arange(s3x)), \
95                           ct['img'].astype('float'), \
96                           np.dstack((pzn,pyn,pxn)), \
97                           method='linear', bounds_error=False, fill_value=0)
98
99     oRayInterp[idx[0],idx[1]] = oRayInterp_i
100

```

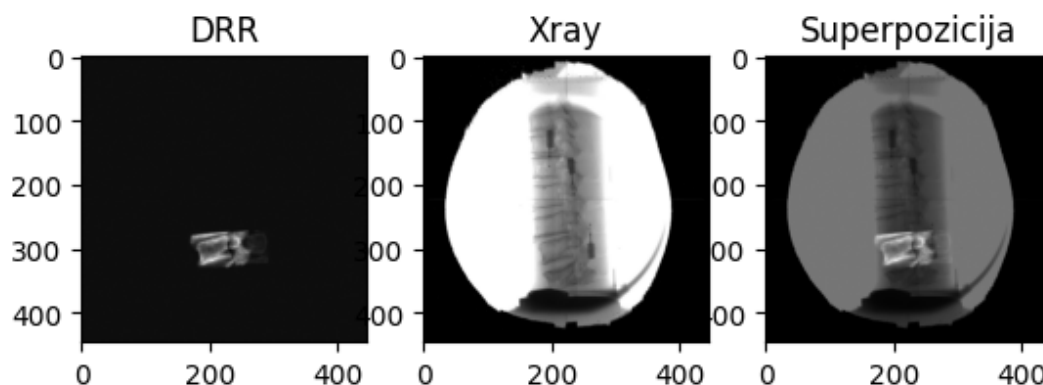
```

101     # izvedi dejansko operacijo vzdolž žarkov
102     oRayInterp = np.mean(oRayInterp, axis=1).reshape(rectShape)
103
104     # ustvari izhodne spremenljivke
105     oDRR = np.zeros_like(Xray['img']).astype('float')
106     oMask = np.zeros_like(Xray['img'])
107
108     oDRR[ymin:ymax, xmin:xmax] = oRayInterp
109     oMask[ymin:ymax, xmin:xmax] = 255
110
111     return oDRR, oMask
112
113 # preizkus funkcije
114 iStep = 3 # korak vzorčenja vzdolž žarkov v mm
115
116 # predobdelava 3D in 2D slik
117 ct_temp = preprocess_ct(ct)
118 Xray_temp = preprocess_Xray(Xray)
119
120 ts = time.perf_counter()
121 oDRR, oDRRMask = rvlib.project3DTo2D(ct_temp, Xray_temp, iStep)
122 print('Čas izračuna: {:.1f} ms'.format(1000.0*(time.perf_counter()-ts)))
123
124 # tri osi v enem prikaznem oknu
125 f, ax = plt.subplots(1, 3)
126 ax[0].imshow(oDRR, cmap='gray')
127 ax[0].set_title('DRR')
128 ax[1].imshow(Xray_temp['img'], cmap='gray')
129 ax[1].set_title('Xray')
130 ax[2].imshow(2*rvlib.normalizeImage(oDRR, iType='range') +
131             rvlib.normalizeImage(Xray_temp['img'], iType='range'), cmap='gray')
132 ax[2].set_title('Superpozicija')
133 plt.show()

```

Slika 9.4 prikazuje rezultat uporabe funkcije, ki se izvede v:

Čas izračuna: 78.3 ms



Slika 9.4: Vizualizacija DRR (*levo*), rentgenske slike (*sredina*) in njunega seštevka (*desno*).

Vaja 9.5 Napišite funkcijo za 3D togo preslikavo vzorčne mreže 3D CT slike okoli njenega središča:

```
1 def rigidTrans( ct, iPar ):
2     # Tu napišite python kodo funkcije
3     return oTrans
```

kjer vhodna spremenljivka *ct* predstavlja 3D CT sliko, *iPar* pa šestvrstični vektor $\mathbf{p} = [t_x, t_y, t_z, \alpha, \beta, \gamma]^T$ s parametri 3D toge preslikave, kjer so rotacije definirane glede na center 3D slike. Izhodna spremenljivka *oTrans* naj predstavlja togo preslikavo 3D CT slike.

Razširite funkcijo za 2D projekcijo oz. DRR sliko tako, da bo imela dodatni vhodni parameter *iPar*, npr. `project3DTo2D( ..., iPar )`, kjer je *iPar* je šestvrstični vektor s parametri 3D toge preslikave 3D CT slike okoli njenega središča. Za 3D togo preslikavo 3D CT slike uporabite funkcijo `rigidTrans( ct, iPar )`. Prikažite 2D projekcijske slike za različne vrednosti parametrov preslikave, npr. *iPar* = [0 20 0 0 0 0], *iPar* = [0 0 0 0 45 0], *iPar* = [0 0 0 0 0 90].

Funkcija `rigidTransMatrix()` generira togo transformacijsko matriko za 3D sliko na podlagi šestih parametrov (*iPar*), ki vključujejo tri translacije (*tx*, *ty*, *tz*) in tri rotacije (*alpha*, *beta*, *gamma*). Postopek vključuje ustvarjanje rotacijske matrike (*oRot*) in translacijske matrike (*oTrans*), ter upoštevanje središča slike (*oCenter* in *oInvCenter*) za pravilno poravnavo transformacije. Funkcija `project3DTo2D()` nato uporabi to matriko za transformacijo koordinatnega sistema CT slike, kjer kombinira obstoječo transformacijsko matriko (*ct['TPos']*) z novo togo transformacijo, in nato izvede konično projekcijo 3D slike na 2D ravnino z uporabo knjižnične funkcije `rvlib.project3DTo2D`, ki upošteva korak vzorčenja (*iStep*) in generira simulirani rentgenski posnetek. Obe funkciji skupaj omogočata projekcijo 3D volumna z možnostjo prilagajanja položaja in orientacije:

```
1 def rigidTransMatrix(ct, iPar): # iPar = [tx, ty, tz, alpha, beta, gamma]
2     s3z,s3y,s3x = ct['img'].shape
3     oRot = rvlib.transAffine3D(iTrans=(0,0,0), iRot=(iPar[3],iPar[4],iPar[5]))
4     oTrans = rvlib.transAffine3D(iTrans=(iPar[0],iPar[1],iPar[2]), iRot=(0,0,0))
5     oCenter = rvlib.transAffine3D(iTrans=(-s3x/2, -s3y/2, -s3z/2), iRot=(0,0,0))
6     oInvCenter = rvlib.transAffine3D(iTrans=(s3x/2, s3y/2, s3z/2), iRot=(0,0,0))
7     return np.dot(oTrans, np.dot(oInvCenter, np.dot(oRot, oCenter)))
8
9 def project3DTo2D(ct, Xray, iStep, iPar=[0,0,0,0,0,0]):
10     TPos = np.dot(ct['TPos'], rigidTransMatrix(ct, iPar))
11     newCt = {'img':ct['img'], 'TPos':TPos}
12     return rvlib.project3DTo2D(newCt, Xray, iStep)
13
14 # preizkus funkcije
15 iPar = [0, 0, -30, -45, 0, 0]
16 iStep = 3
17
18 # predobdelava 3D in 2D slik
19 ct_temp = preprocess_ct(ct)
20 Xray_temp = preprocess_Xray(Xray)
21
22 ts = time.perf_counter()
23 oDRR, oDRRMask = project3DTo2D(ct_temp, Xray_temp, iStep, iPar)
24 print('Čas izračuna: {:.1f} ms'.format(1000.0*(time.perf_counter()-ts)))
25
```

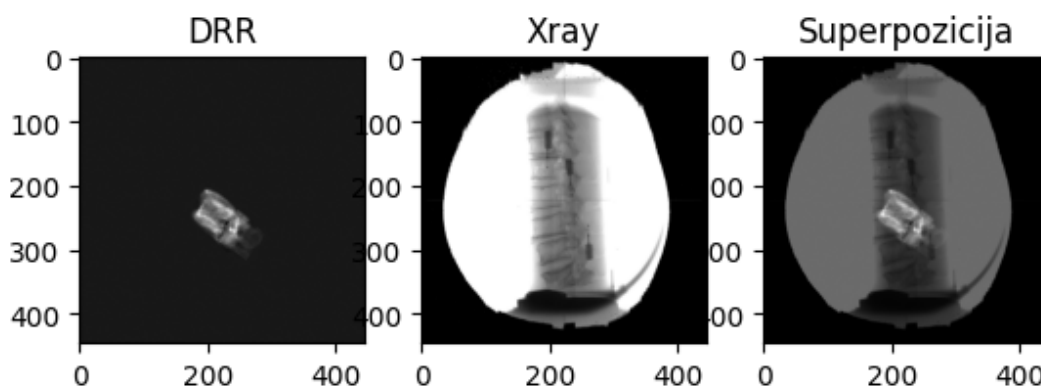


```

26 # tri osi v enem prikaznem oknu
27 f, ax = plt.subplots(1, 3)
28 ax[0].imshow(oDRR, cmap='gray')
29 ax[0].set_title('DRR')
30 ax[1].imshow(Xray_temp['img'], cmap='gray')
31 ax[1].set_title('Xray')
32 ax[2].imshow(2*rvlib.normalizeImage(oDRR, iType='range') +
33             rvlib.normalizeImage(Xray_temp['img'], iType='range'), cmap='gray')
34 ax[2].set_title('Superpozicija')
35 plt.show()

```

Slika 9.5 prikazuje rezultat uporabe funkcije pri parametrih toge preslikave $(t_x, t_y, t_z, \alpha, \beta, \gamma) = [0\text{mm}, 0\text{mm}, -30\text{mm}, -45\text{deg}, 0\text{deg}, 0\text{deg}]$.



Slika 9.5: Vizualizacija DRR (*levo*) pri parametrih toge preslikave $(t_x, t_y, t_z, \alpha, \beta, \gamma) = [0\text{mm}, 0\text{mm}, -30\text{mm}, -45\text{deg}, 0\text{deg}, 0\text{deg}]$, rentgenske slike (*sredina*) in njenega seštevka (*desno*).

Vaja 9.6 Napišite funkcijo, ki izračuna vrednost medsebojne informacije med dvema slikama:

```

1 def mutualInformation( iImageI, iImageJ, iBins ):
2     # Tu napišite python kodo funkcije
3     return oMI

```

kjer sta *iImageI* in *iImageJ* 2D sivinski sliki, *iBins* pa število intervalov za izračun histograma. Funkcija vrne skalar *oMI*. Preizkusite delovanje funkcije tako, da izračunate medsebojno informacijo med projicirano sliko in 2D rentgensko sliko, pri tem pa uporabite le sivinske vrednosti na področju maske projicirane slike. Preučite vpliv števila intervalov *iBins* na vrednost medsebojne informacije.

Izračunajte in izrišite vrednosti *oMI* med 2D projekcijami in 2D rentgensko sliko tako, da spreminjate le po en parameter toge preslikave naenkrat. Translacije t_x , t_y in t_z spreminjajte od -20 do 20 milimetrov s korakom 2 milimetra, rotacije α , β in γ pa od -20 do 20° s korakom 2° in narišite ter ustrezno označite grafe $MI(t_x)$, $MI(t_y)$, $MI(t_z)$, $MI(\alpha)$, $MI(\beta)$, $MI(\gamma)$. ■

Funkcija `mutualInformation()` izračuna medsebojno informacijo med dvema 2D slikama (*iImageI* in *iImageJ*), ki meri statistično odvisnost med njunimi sivinskimi vrednostmi. Ključni koraki vključujejo: (1) pretvorbo sivinskih vrednosti v indekse histograma z `getIndices()`, ki upošteva podano območje *iSpan* (min, max) in število predalov *iBins*; (2) izračun 1D histogramov posameznih slik (`hist1D()`) in 2D skupnega histograma (`hist2D()`) z uporabo konvolucijskega

glajenja za zmanjšanje šuma; (3) normalizacijo histogramov v verjetnostne porazdelitve (p_I , p_J , p_{IJ}); ter (4) izračun entropij (H_I , H_J , H_{IJ}) in medsebojne informacije kot $oMI = H_I + H_J - H_{IJ}$. Glavni parameter $iSpan$ omogoča nastavitve območja sivinskih vrednosti, medtem ko $iBins$ določa granularnost histograma. Rešitev zapišemo:

```

1 def mutualInformation(iImageI, iImageJ, iBins, iSpan=(None, None)):
2     '''
3     Izracunaj medsebojno informacijo med 2D slikama
4
5     Parameters
6     -----
7     iImageI : numpy.ndarray
8         Sivinska informacija slike I
9     iImageJ : numpy.ndarray
10        Sivinska informacija slike J
11     iBins : int
12        Stevilo predalov v histogramu
13     iSpan : tuple | list
14        Obmocje vrednosti (min, max)
15
16     Returns
17     -----
18     oDRR : numpy.ndarray
19        Izhodna iz 3D v 2D prostor preslikana slika, ki simulira rentgen
20     oMask : numpy.ndarray
21        Maska področja kamor se preslika 3D slika (1-ospredje, 0-ozadje)
22     '''
23     iImageI = np.asarray(iImageI)
24     iImageJ = np.asarray(iImageJ)
25
26     # funkcija za pretvorbo sivinskih vrednosti v indekse
27     def getIndices(iData, iBins, iSpan):
28         # doloci obmocje sivinskih vrednosti
29         minVal, maxVal = iSpan
30         maxVal = (np.max(iData)+1e-7 if maxVal is None else maxVal)
31         minVal = (np.min(iData) if minVal is None else minVal)
32         # pretvori v indeks polja
33         idx = np.round((iData - minVal) / (maxVal - minVal) * (iBins-1))
34         idx[idx < 0] = 0
35         idx[idx >= iBins] = iBins-1
36         # vrni indekse
37         return idx.astype('uint32')
38
39     # funkcija za izracun 1D histograma
40     def hist1D(iData, iBins, iSpan):
41         # pretvorba sivinskih vrednosti v indekse
42         idx = getIndices(iData, iBins, iSpan)
43         # izracunaj histogram
44         histData = np.zeros((iBins,))
45         for i in idx:
46             histData[i] += 1
47         # vrni histogram
48         # return histData
49         # vrni glajeni histogram
50         return ni.convolve(histData, np.array([0.2, 0.6, 0.2]))
51

```

```

52     # funkcija za izracun 2D histograma
53     def hist2D(iData1, iData2, iBins, iSpan):
54         # pretvorba sivinskih vrednosti v indekse
55         idx1 = getIndices(iData1, iBins, iSpan)
56         idx2 = getIndices(iData2, iBins, iSpan)
57         # izracunaj histogram
58         histData = np.zeros((iBins, iBins))
59         for (i, j) in zip(idx1, idx2):
60             histData[i,j] += 1
61         # vrni histogram
62         # return histData
63         # vrni glajeni histogram
64         return ni.convolve(histData, np.array([
65             [1, 2, 1], [2, 8, 2], [1, 2, 1]])/20.0)
66
67     # izracunaj histograme slik
68     hI = hist1D(iImageI, iBins, iSpan)
69     hJ = hist1D(iImageJ, iBins, iSpan)
70     hIJ = hist2D(iImageI, iImageJ, iBins, iSpan)
71
72     # normaliziraj histograme v gostote verjetnosti
73     pI = hI / (np.sum(hI) + 1e-7)
74     pJ = hJ / (np.sum(hJ) + 1e-7)
75     pIJ = hIJ / (np.sum(hIJ) + 1e-7)
76
77     # izracunaj entropije
78     HI = np.sum(- pI[pI>0] * np.log(pI[pI>0]))
79     HJ = np.sum(- pJ[pJ>0] * np.log(pJ[pJ>0]))
80     HIJ = np.sum(- pIJ[pIJ>0] * np.log(pIJ[pIJ>0]))
81
82     # izracunaj medsebojno informacijo
83     oMI = HI + HJ - HIJ
84
85     return oMI

```

Rezultat oMI je skalarna vrednost, ki kvantificira podobnost slik, pri čemer višje vrednosti nakazujejo močnejšo statistično povezanost. Funkcija je uporabna pri poravnavi slik, kjer se maksimizacija medsebojne informacije uporablja kot kriterij v numerični optimizaciji. Testno kodo za izračun vrednosti medsebojne informacije glede na spremembo posameznih parametrov toge preslikave zapišemo kot:

```

1  # parametri
2  iBins = 32
3  iSpan = [0.0, 255.0]
4
5  # predobdelava 3D in 2D slik
6  ct_temp = preprocess_ct(ct)
7  Xray_temp = preprocess_Xray(Xray)
8
9  par_range = np.arange(-20, 20, 2)
10 oMI_plot = np.zeros_like(par_range).astype('float')
11
12 # normalizacija intenzitet v projekciji
13 oDRR, oDRRMask = project3DTo2D(ct_temp, Xray_temp, iStep)
14 ids = np.where(oDRRMask>0)

```

```

15 xmin, xmax = np.min(ids[1]), np.max(ids[1])
16 ymin, ymax = np.min(ids[0]), np.max(ids[0])
17 iTemp = oDRR[ymin:ymax,xmin:xmax]
18 imin, imax = np.min(iTemp), np.max(iTemp)
19
20 for idx in range(len(par_range)):
21     p = par_range[idx]
22     oDRR, oDRRMask = project3DTo2D(ct_temp, Xray_temp, iStep,
23                                   iPar=[p,0,0,0,0,0])
24     # poklici funkcijo
25     ids = np.where(oDRRMask>0)
26     xmin, xmax = np.min(ids[1]), np.max(ids[1])
27     ymin, ymax = np.min(ids[0]), np.max(ids[0])
28     iImageI = Xray_temp['img'][ymin:ymax,xmin:xmax]
29     iImageJ = (oDRR[ymin:ymax,xmin:xmax] - imin) / (imax - imin) * 255.0
30
31     oMI = rvlib.mutualInformation(iImageI, iImageJ, iBins, iSpan)
32     oMI_plot[idx] = oMI
33
34     print('p: {:.1f}, MI={:.2f}'.format(p, oMI))
35
36 plt.close('all')
37 plt.plot(par_range, oMI_plot)
38 plt.suptitle('MI(t_x)')
39 plt.xlabel('t_x')
40 plt.ylabel('MI')

```

Rezultat prikazuje slika 9.6. Pri vrednostih parametrov $iPar = [0, 0, 0, 0, 0, 0]$ sta 3D in 2D sliki ustrezno poravnavi glede na ročno referenčno poravnavo, kar pomeni, da pričakujemo v tej legi maksimum medsebojne informacije, ob vsakem odkliku od te lege pa zmanjšanje vrednosti. Opazimo, da je za parametra t_x in t_y medsebojna informacija brez izrazitega vrha, kar je posledica tega, da omenjena parametra delno kodirata translacijo v smeri projekcije – premiki v tej smeri komaj opazno skalirajo DRR, zato je občutljivost medsebojne informacije na te premike nizka.

Vaja 9.7 Načrtajte avtomatski postopek za poravnavo 3D in 2D slik tako, da ustvarite kriterijsko funkcijo:

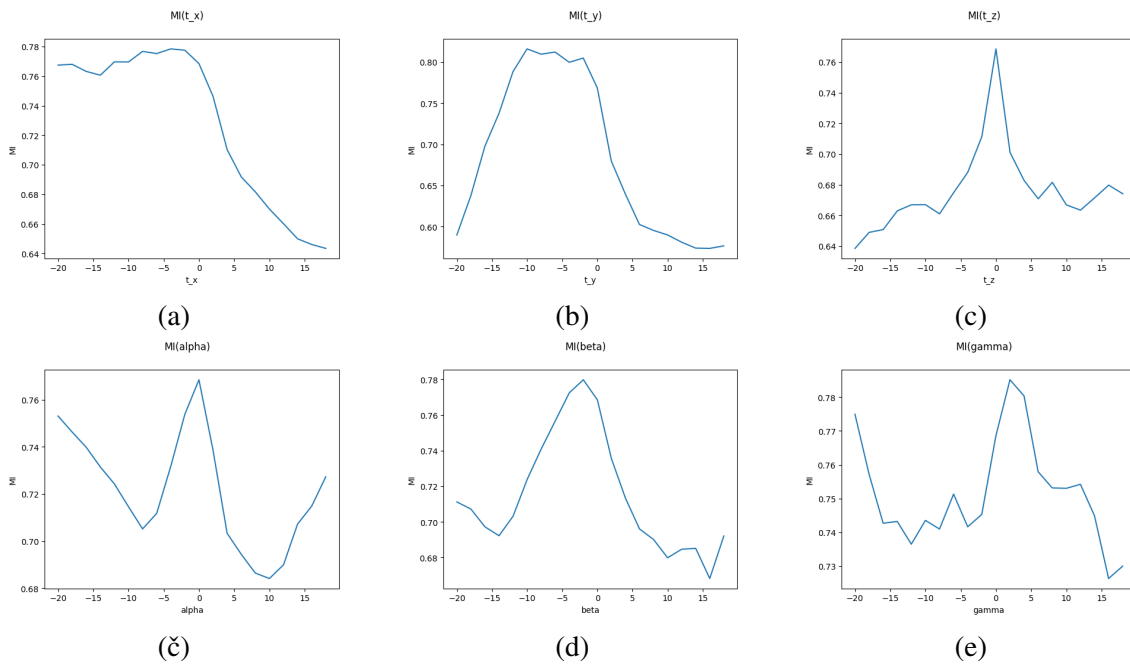
```

1 def criterionFcn( iPar, ct, Xray ):
2     # Tu napišite python kodo funkcije
3     return oMP

```

ki pri podanih parametrih 3D toge preslikave $iPar$ izračuna vrednost medsebojne informacije MI med DRR in 2D rentgensko sliko. Optimalne vrednosti parametrov $\mathbf{p}^*$ toge preslikave določite s pomočjo simpleksne ali katere druge metode optimizacije (vaja 8). ■

Kriterijska funkcija `criterionFcn()` je zasnovana za optimizacijo poravnave med 3D CT sliko (`ct`) in 2D rentgenskim posnetkom (`Xray`) z uporabo medsebojne informacije (MI) kot mere podobnosti. Funkcija najprej generira simulirani rentgenski posnetek (DRR) iz 3D volumna s funkcijo `project3DTo2D()` za dane transformacijske parametre $iPar$ (translacija in rotacija). Nato izračuna medsebojno informacijo med realnim rentgenskim posnetkom in simuliranim DRR-jem z uporabo funkcije `mutualInformation()`, pri čemer se upošteva le območje, kjer je maska `oDRRMask` aktivna. Vrednosti DRR-ja se normalizirajo v območje 0-255, kriterijska funkcija pa



Slika 9.6: Potek vrednosti medsebojne informacije glede na spremembo parametrov toge preslikave, in sicer translacije (a) t_x , (b) t_y , (c) t_z , in rotacije (č) α , (d) β in (e) γ .

vrne negativno vrednost MI, saj optimizacijski postopek minimizira ciljno funkcijo. Kriterijsko funkcijo zapišemo:

```
1 def criterionFcn(iPar, ct, Xray):
2     iStep = 3
3     iBins = 32
4     iSpan = [0.0, 255.0]
5     imin = -0.80841250322681901
6     imax = 13.620590454314147
7
8     oDRR, oDRRMask = project3DTo2D(ct, Xray, iStep, iPar)
9
10    ids = np.where(oDRRMask>0)
11    xmin, xmax = np.min(ids[1]), np.max(ids[1])
12    ymin, ymax = np.min(ids[0]), np.max(ids[0])
13    iImageI = Xray['img'][ymin:ymax,xmin:xmax]
14    iImageJ = oDRR[ymin:ymax,xmin:xmax]
15    iImageJ = (oDRR[ymin:ymax,xmin:xmax] - imin) / (imax - imin) * 255.0
16
17    return -rvlib.mutualInformation(iImageI, iImageJ, iBins, iSpan)
```

V eksperimentu za numerično optimizacijo uporabimo Powellovo metodo za optimizacijo rotacijskega parametra α , kjer vsaka iteracija vključuje klic kriterijske funkcije in vizualizacijo trenutnega stanja s primerjavo originalnega rentgenskega posnetka, DRR-ja pred in po poravnavi ter njihove superpozicije. Glavni cilj je maksimizirati medsebojno informacijo, kar pomeni najboljšo možno ujemanje med simuliranim in realnim posnetkom:

```
1 # definiramo začetni odmik
2 iAlphaStart = -8
3
```

```

4 # predobdelava 3D in 2D slik
5 ct_temp = preprocess_ct(ct)
6 Xray_temp = preprocess_Xray(Xray)
7
8 # definicija funkcije MP(p)
9 oMP = lambda iPar : criterionFcn([0,0,0,float(iPar),0,0], ct_temp, Xray_temp)
10
11 neval = 1
12 def callbackF(iPar):
13     global neval
14     fval = criterionFcn([0,0,0,float(iPar),0,0], ct_temp, Xray_temp)
15     print('{0:4d}   {1: 3.6f}   {2: 3.6f}'.format(neval, iPar[0], fval))
16     neval += 1
17 print('{0:4s}   {1:9s}   {2:9s}'.format('Iter', ' Alpha', ' MI(Alpha)'))
18
19 # klic optimizacije
20 ts = time.perf_counter()
21 # simpleksna optimizacija
22 # iAlphaEnd = fmin(func=oMP, x0=iAlphaStart, maxiter=100, xtol=1e-5, ftol=1e-5,
23     disp=1)[0]
24 # method = Nelder-Mead (simpleksna optimizacija), Powell
25 res = minimize(fun=oMP, x0=iAlphaStart, method='Powell', tol=1e-5,
26     options={'maxiter':100, 'maxfev':1000, 'xtol':1e-5,
27     'ftol':1e-5, 'disp':True}, callback=callbackF
28 )
29 iAlphaEnd = float(res.x)
30 print('Čas izračuna: {:.1f} s'.format(time.perf_counter()-ts))
31 print('Alpha pred poravnavo: {:.1f} deg\nAlpha po poravnavi: {:.1f} deg'.format(
32     iAlphaStart, iAlphaEnd)
33 )
34
35 iParStart = [0, 0, 0, iAlphaStart, 0, 0]
36 iParEnd = [0, 0, 0, iAlphaEnd, 0, 0]
37
38 # prikaz rezultatov
39 oDRR_start, oDRRMask = project3DTo2D(ct_temp, Xray_temp, 3, iParStart)
40 oDRR_end, oDRRMask = project3DTo2D(ct_temp, Xray_temp, 3, iParEnd)
41
42 # šest osi v enem prikaznem oknu
43 f, ax = plt.subplots(2, 3)
44 ax[0, 0].imshow(Xray_temp['img'], cmap='gray')
45 ax[0, 0].set_title('Xray')
46 ax[1, 0].imshow(Xray_temp['img'], cmap='gray')
47
48 ax[0, 1].imshow(oDRR_start, cmap='gray')
49 ax[0, 1].set_title('DRR pred poravnavo')
50
51 ax[1, 1].imshow(oDRR_end, cmap='gray')
52 ax[1, 1].set_title('DRR po poravnavi')
53
54 ax[0, 2].imshow(2*rvlib.normalizeImage(oDRR_start, iType='range') +
55     rvlib.normalizeImage(Xray_temp['img'], iType='range'), cmap='gray')
56 ax[0, 2].set_title('Superpozicija')
57
58 ax[1, 2].imshow(2*rvlib.normalizeImage(oDRR_end, iType='range') +
59     rvlib.normalizeImage(Xray_temp['img'], iType='range'), cmap='gray')

```



```
59 ax[1, 2].set_title('Superpozicija')
60 plt.show()
```

Izpis v ukazni vrstici je naslednji:

```
Iter    Alpha    MI(Alpha)
<ipython-input-24-c905437b8b05>:29: DeprecationWarning: Conversion of an array
    ↳ with ndim > 0 to a scalar is deprecated, and will error in future. Ensure
    ↳ you extract a single element from your array before performing this
    ↳ operation. (Deprecated NumPy 1.25.)
    oMP = lambda iPar : criterionFcn([0,0,0,float(iPar),0,0], ct_temp, Xray_temp)
<ipython-input-24-c905437b8b05>:34: DeprecationWarning: Conversion of an array
    ↳ with ndim > 0 to a scalar is deprecated, and will error in future. Ensure
    ↳ you extract a single element from your array before performing this
    ↳ operation. (Deprecated NumPy 1.25.)
fval = criterionFcn([0,0,0,0,float(iPar),0], ct_temp, Xray_temp)
1   -5.368449  -0.760157
2   -0.591433  -0.778438
3   -0.599876  -0.776393
4   -0.603923  -0.777245
5   -0.086603  -0.766410
6   -0.092069  -0.766163
7   -0.092069  -0.766163
Optimization terminated successfully.
    Current function value: -0.772473
    Iterations: 7
    Function evaluations: 204
Čas izračuna: 14.0 s
Alpha pred poravnavo: -8.0 deg
Alpha po poravnavi: -0.1 deg
```

Z metodo optimizacije smo uspešno kompenzirali začetni premik $\alpha = -8\text{deg}$ v končni odklik $\alpha = -0,1\text{deg}$, kar je zanemarljivo majhna napaka saj je referenčna poravnava pri $\alpha = -0\text{deg}$. Prikaz situacije pred in po poravnavi je na sliki 9.7.

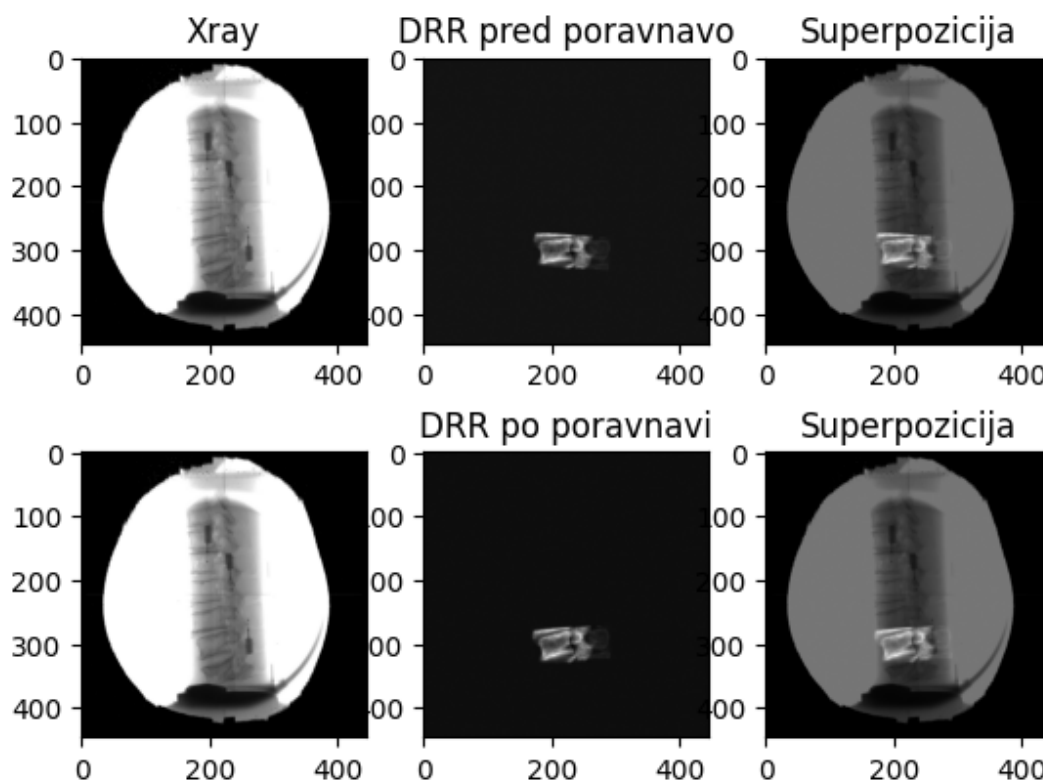
9.5 Naloge in vprašanja

Poročilo naloge pripravite v obliki Python skripte, ki naj izvede zahtevane izračune oz. klice funkcij in izriše pripadajoče slike oz. grafe. Vaša dolžnost je, da v skripti uvozite ustrezne Python knjižnice in dodate morebitne podporne skripte. Skripta se mora izvesti in poustvariti rezultate, kot se zahtevajo v nalogah. Na vprašanja odgovorite v obliki komentarja, naprimer *# Naloga X.1: To je moj odgovor..*

Naloga 9.1 Z linearnim oknjenjem 2D rentgenske slike lahko izboljšate kontrast na področju vretenc. Preslikajte nepredznačene 16-bitne sivinske vrednosti 2D rentgenske slike v nepredznačene 8-bitne sivinske vrednosti na območju tako, da se na izhodni sliki poveča kontrast na področju vretenc. *Sivinsko preslikano 2D rentgensko sliko uporabite pri vseh naslednjih nalogah!*

Naloga 9.2 Vsaki od 3D CT slik določite mediano sivinskih vrednosti in jo odštejte od vseh sivinskih vrednosti. Pred odštevanjem ustrezno pretvorite sivinske vrednosti. Prikažite medialne oz. sredinske 2D rezine v stranskem, čelnem in prečnem pogledu. *Sivinsko preslikani 3D CT slike uporabite pri vseh naslednjih nalogah!*

Naloga 9.3 Izračunajte in izrišite vrednosti oMI med 2D projekcijami in 2D rentgensko sliko tako, da spreminjate le po en parameter toge preslikave naenkrat. Translacije t_x , t_y in t_z spreminjajte od



Slika 9.7: Vizualizacija DRR (*levo*), rentgenske slike (*sredina*) in njunega seštevka (*desno*) pred (*zgoraj*) in po poravnavi (*zgoraj*).

-20 do 20 milimetrov s korakom 2 milimetra, rotacije α , β in γ pa od -10 do 10° s korakom 1° in narišite ter ustrezno označite grafe $MI(t_x)$, $MI(t_y)$, $MI(t_z)$, $MI(\alpha)$, $MI(\beta)$, $MI(\gamma)$.

Naloga 9.4 Obrazložite povezavo med medsebojno informacijo in dejansko podobnostjo 2D rentgenske in 2D projekcijskimi slikami. Katere vrednosti medsebojne informacije, višje ali nižje, odražajo dejansko podobnost med 2D rentgensko in 2D projekcijskimi oz. DRR slikami? Kolikšni sta teoretično minimalna in maksimalna vrednost medsebojne informacije?

Naloga 9.5 Optimalne vrednosti parametrov $\mathbf{p}^*$ toge preslikave dobimo s klicem simpleksne minimizacije:

```
1 # definicija funkcije MP(p)
2 oMP = lambda iPar : criterionFcn( iPar, ct, Xray )
3 # klic simpleksne optimizacije
4 from scipy.optimize import fmin
5 iPar_opt = fmin( func=oMP, x=np.zeros(6,1), maxiter=200, xtol=1e-5, ftol=1e-5,
    disp=1 )
```

Naloga 9.6 Izvedite poravnavo slike ct2 in Xray. Navedite vrednosti optimalnih parametrov $\mathbf{p}^*$ toge preslikave, ki rezultirajo v uspešni poravnavi 3D in 2D slik, in prikažite pripadajočo DRR sliko. Uspešnost postopka poravnave lahko preverite kot je prikazano na sliki 9.8.

Naloga 9.7 Načrtajte poravnavo 3D CT slike ct2 in 2D rentgenske slike Xray s pomočjo ročno določenih korespondenčnih točk. Zapišite model preslikave med slikama in parametre preslikave

in določite model kamere, pri čemer uporabite podatke o položaju izvora in legi 2D detektorja. Optimizacijo za iskanje optimalnih parametrov izvedite na več načinov:

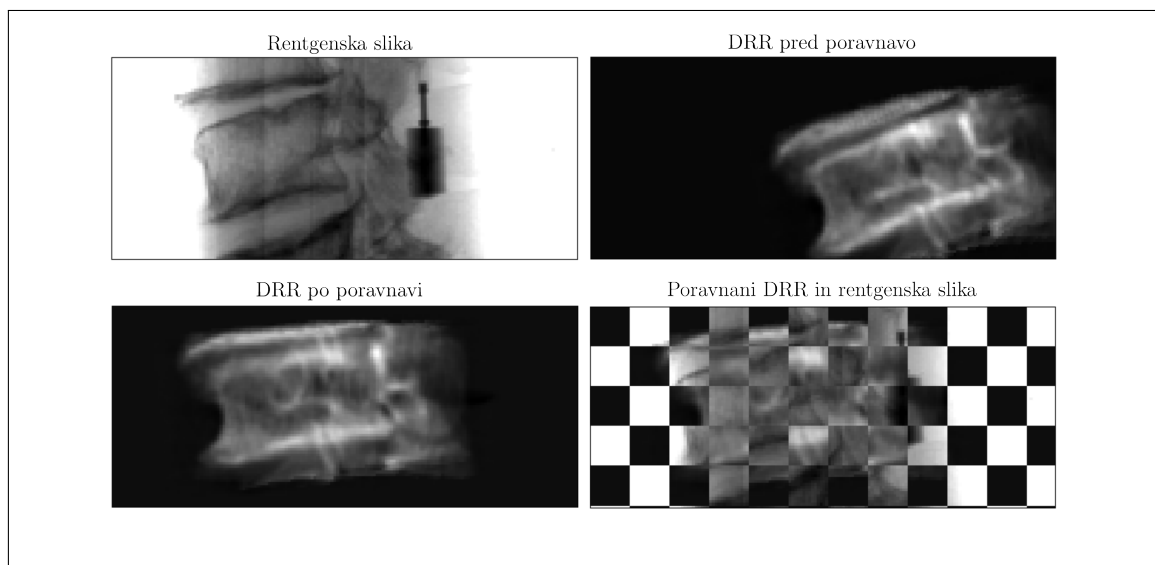
- Z *direktno linearno transformacijo* (DLT).
- Z nelinearno optimizacijo, pri čemer linearizirate zaprtozančni izračun koraka s $\mathbf{p} + \Delta\mathbf{p}$ in izpeljete enačbe za izračun $\Delta\mathbf{p}$.
- Kombinacija DLT in nato nelinearna optimizacija.

Kateri način iskanja vrne najboljšo poravnavo 3D in 2D slik?

Naloga 9.8 Določite napako poravnave 3D CT slike ct2 in 2D rentgenske slike Xray iz prejšnjih nalog tako, da kot referenčno poravnavo uporabite podatke o legi 3D CT slike ct. Kot referenčne točke uporabite oglišča 3D slike. Izračunajte napako na dva načina:

- Napaka v 3D prostoru (*ang. target registration error: TRE*) kot srednjo kvadratno napako med korespondenčnimi točkami oglišč 3D slike v referenčni legi in trenutni legi.
- Napaka v ravnini detektorja (*ang. projected registration error: PRE*) kot srednjo kvadratno napako med korespondenčnimi točkami oglišč 3D slike v referenčni legi in trenutni legi, pri točke predhodno projicirate v 2D ravnino.

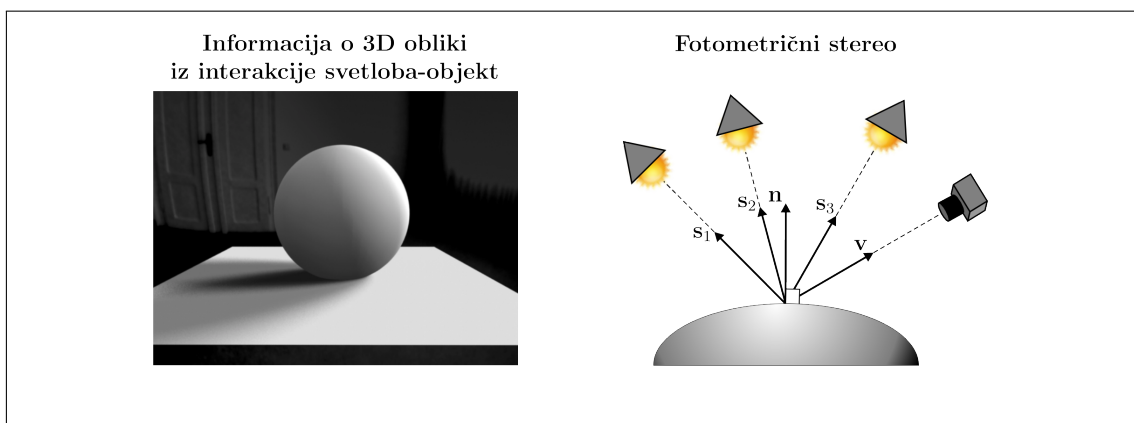
Rezultati nekaterih nalog:



Slika 9.8: Povečan prikaz poravnave vretenca pred in po poravnavi CT in rentgenskega posnetka. Poza CT je prikazana z vizualizacijo DRR, ki mora ob uspešni poravnavi sovpadati z rentgenskim posnetkom. Ujemanje robov je razvidno iz mešanega prikaza rentgena in DRR.

10. Rekonstrukcija 3D oblik

Pri rekonstrukciji 3D oblik želimo s pomočjo ene ali več 2D slik nekega objekta, zajetih iz različnih pogledov in/ali pod različnimi osvetlitvami, najti optimalno 3D obliko tega objekta tako, da bo 3D oblika čim boljje sovpadala z zajetimi 2D slikami. Eden od postopkov rekonstrukcije 3D oblik je **fotometrični stereo** (slika 10.1), pri katerem izluščimo informacijo o obliki na podlagi senc na objektu. Sence ustvarimo z osvetlitvijo objekta iz različnih smeri (npr. s_1, s_2, s_3), slike pa zajamemo iz enega pogleda (v). Iz zajetih slik dobimo v vsaki točki slike (x, y) oceno normale na površino n , z integriranjem teh normal glede na neko izhodiščno točko dobimo rekonstrukcijo 3D oblike.



Slika 10.1: Interakcija svetloba-objekt na podlagi senc razkrije obliko objekta (*levo*), kar za rekonstrukcijo 3D oblike izkorišča tehnika fotometrični stereo (*desno*).

10.1 Fotometrični stereo in Lambertov model

Če predpostavimo, da pri interakciji svetloba-površina pride le do difuznega odboja, potem lahko enostavno povežemo svetilnost slikovnega elementa z obliko površine. Uporabimo **Lambertov**

model difuzne površine:

$$I(x, y) = \frac{kc}{\pi} \rho_i \cos \theta_i = \frac{kc}{\pi} \rho_i s \circ n,$$

kjer je k svetilnost vira, c konstanta optičnega sistema, ρ albedo oz. odbojnost površine. Če predpostavimo, da je $\frac{kc}{\pi} = 1$ potem dobimo za j -to svetilo ($j = 1, \dots, N$) enačbo za svetilnost slikovnega elementa v obliki $I_j(x, y) = \rho_i \cos \theta_i = \rho_i s_j \circ n$.

10.2 Izračun albeda in normal na površino objekta

Za oceno normale na površino n v točki slike (x, y) potrebujemo vsaj tri svetila ($N = 3$) iz različnih smeri (s_1, s_2, s_3) , da lahko zapišemo minimalen sistem linearnih enačb.

Sistem enačb za tri svetila:

$$\begin{aligned} I_1 &= \rho s_1^T \cdot n \\ I_2 &= \rho s_2^T \cdot n \\ I_3 &= \rho s_3^T \cdot n \end{aligned}$$

Sistem enačb v matrični obliki:

$$\Rightarrow \underbrace{\begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix}}_{I_{(3 \times 1)}} = \underbrace{\begin{bmatrix} s_1^T \\ s_2^T \\ s_3^T \end{bmatrix}}_{S_{(3 \times 3)}} \underbrace{\rho n}_{\tilde{n}_{(3 \times 1)}}$$

Rešitev sistema:

$$\begin{aligned} \tilde{n} &= S^{-1} I \\ \rho &= \|\tilde{n}\| \\ n &= \tilde{n} / \|\tilde{n}\| = \|\tilde{n}\| / \rho \end{aligned}$$

Z gornjim postopkom dobimo oceno normale n na površino objekta v vsaki točki slike (x, y) . Ocena normale je bolj robustna, če uporabimo več osvetlitev objekta iz več med seboj različnih smeri s_j , $N > 3$. V katerem od j pogledov so lahko nekateri slikovni elementi relativno temni, zato bo ocena normale v teh slikovnih elementih manj zanesljiva, kar rešimo z uteževanjem enačb s svetlostjo $I_j = I_j(x, y)$ iz slik posameznega pogleda s_j .

Sistem enačb za N svetil:

$$\begin{aligned} I_1^2 &= I_1 (\rho s_1^T \cdot n) \\ &\vdots \\ I_N^2 &= I_N (\rho s_N^T \cdot n) \end{aligned}$$

Sistem enačb v matrični obliki:

$$\Rightarrow \underbrace{\begin{bmatrix} I_1^2 \\ \vdots \\ I_N^2 \end{bmatrix}}_{I_{(N \times 1)}} = \underbrace{\begin{bmatrix} I_1 s_1^T \\ \vdots \\ I_N s_N^T \end{bmatrix}}_{S_{(N \times 3)}} \underbrace{\rho n}_{\tilde{n}_{(3 \times 1)}}$$

Rešitev sistema:

$$\begin{aligned} I &= S \tilde{n} \\ S^T I &= S^T S \tilde{n} \\ \tilde{n} &= (S^T S)^{-1} S^T I \end{aligned}$$

10.3 Rekonstrukcija 3D oblike

Sistem enačb za določanje normale na površino rešimo za vsako točko (x, y) in tako dobimo sliko normal na površino $n(x, y)$, s pomočjo katere lahko na več načinov rekonstruiramo 3D obliko objekta kot sliko površine oz. globine $z(x, y)$. Najenostavnejši način je, da integriramo sliko normal $n = [n_x, n_y, n_z]^T$ po poljubni krivulji od izhodišča (x_0, y_0) do izbrane točke (x, y) :

$$z(x, y) = \int_{x_0}^x \frac{n_x(u, y)}{n_z(u, y)} du + \int_{y_0}^y \frac{n_x(x, v)}{n_z(x, v)} dv$$

Sliko površine $z(x, y)$ naprimer lahko določimo z analizo odvodov na površino $\partial z / \partial x$ in $\partial z / \partial y$, ki morata biti v vsaki točki (x, y) pravokotna na normalo n (slika 10.2). Iz pripadajočih enačb za sliko velikosti XY zgradimo linearen sistem $Mz = p$, katerega rešitev je slika površine $z(x, y)$.

Matrika M ima dimenzije $(2XY \times XY)$, vektor p pa $(2XY \times 1)$. Matrika M je velika, a redka matrika (večina vrednosti je enaka 0), zato za manjšo obremenitev delovnega spomina uporabimo knjižnico `scipy.sparse` za definicijo redke matrike s funkcijo `csc_matrix()`, vrednosti pa v matriko vnašamo posamezno z naslavljanjem vrstic in stolpcev. Rešitev sistema dobimo s funkcijo `lsqr()` v knjižnici `scipy.sparse.linalg`.

Parcialna odvoda na površino sta vektorja pravokotna na normalo n :

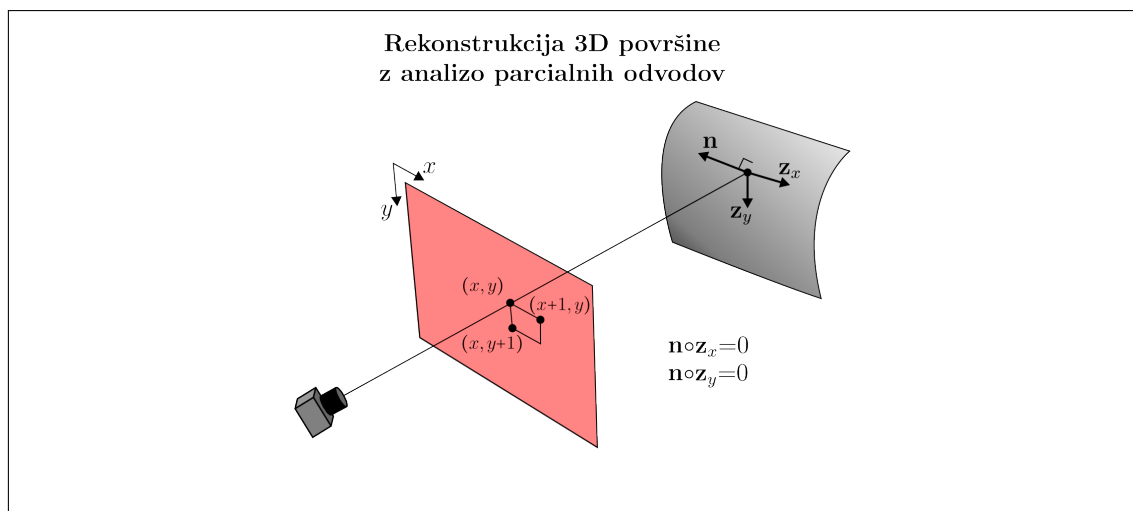
$$\begin{aligned} z_x &= \partial z / \partial x \\ z_y &= \partial z / \partial y \\ \rightarrow n \circ z_x &= 0 \\ \rightarrow n \circ z_y &= 0 \end{aligned}$$

Parcialne odvode izračunamo s končnimi diferencami in za vsako točko (x, y) dobimo dve linearni enačbi:

$$\begin{aligned} z_x &= (x+1, y, z(x+1, y)) - (x, y, z(x, y)) \\ &= (1, 0, z(x+1, y) - z(x, y)) \\ z_y &= (x, y+1, z(x, y+1)) - (x, y, z(x, y)) \\ &= (0, 1, z(x, y+1) - z(x, y)) \\ \rightarrow n_x + n_z(z(x+1, y) - z(x, y)) &= 0 \\ \rightarrow n_y + n_z(z(x, y+1) - z(x, y)) &= 0 \end{aligned}$$

Združimo linearne enačbe za vse točke (x, y) in dobimo rešitev sistema:

$$\begin{aligned} Mz &= p \\ M &\in \{0, n_z, -n_z\} \\ p &\in \{n_x, n_y\} \\ \rightarrow z &= (M^T M)^{-1} M^T p \end{aligned}$$



Slika 10.2: Rekonstrukcija 3D oblike z analizo parcialnih odvodov, ki naj bodo pravokotni na normalo površine – slednji pogoj omogoča zasnovo linearnega predoločenega sistema enačb, katerega rešitev vrne 3D obliko.

10.4 Vaje z rešitvami

Gradivo za vajo v `owl.zip` vsebuje datoteko `lights.txt`, v kateri so podane smeri svetil glede na objekt, slike objekta `owl.k.tif`, pri čemer je $k = 0, \dots, 11$, in pripadajočo masko objekta `owl.mask.tif`. Koda vaje je dostopna na [Git repozitoriju](#).

Najprej uvozimo Python knjižnice:

```
1 import scipy.ndimage as ni
2 import matplotlib.pyplot as plt
3 import matplotlib.cm as cm
4 import numpy as np
5 import scipy as sp
6 from scipy.interpolate import interpn
```

```
7 | from rvlib import *
```

Vaja 10.1 Napišite funkcijo za branje tekstovne datoteke s smermi svetil:

```
1 | def readTxt( iFileName ):
2 |     # Tu napišite python kodo funkcije
3 |     return oLightDir
```

kjer parameter `iFileName` predstavlja ime datoteke za branje ('lights.txt'). Datoteko odprete in zaprete s funkcijama `open()` in `close()`, posamezne vrstice pa preberete s funkcijo `readline()`. Za pretvorbo števil iz besedila v numerični tip lahko uporabite funkciji `split()` in `float()`. Funkcija vrne matriko `oLightDir`, ki bo imela za dano datoteko 'lights.txt' dimenzije 12×3 . ■

Funkcija `readTxt(iFileName)` naj v Pythonu prebere tekstovno datoteko s podanim imenom `iFileName`, kjer vsaka vrstica vsebuje tri številske vrednosti (ločene s presledki), jih pretvori v matriko s tremi stolpci in jih shrani v numpy array `oLightsDir`; najprej inicializira prazno polje tipa `numpy.array`, nato bere datoteko vrstico po vrstico, vsako vrstico razdeli na posamezne številke (kot `float` zapis), jih preoblikuje v vrstico matrike (1×3) in jih vertikalno dodaja v obstoječo matriko (`np.vstack`), dokler ne prebere vseh vrstic, na koncu pa vrne končno matriko smeri svetil v obliki numpy arraya dimenzije $N \times 3$, kjer N predstavlja število prebranih vrstic. Rešitev lahko zapišemo:

```
1 | def readTxt( iFileName ):
2 |     # naloži smeri svetil
3 |     oLightsDir = np.array([])
4 |     with open( iFileName, "r" ) as f:
5 |         while True:
6 |             line = f.readline()
7 |             if not line:
8 |                 break
9 |             # preberi in preoblikuj vrednosti
10 |            numbers = [float(x) for x in line.split()]
11 |            numbers = np.array(numbers).reshape( (1,3) )
12 |            if oLightsDir.size == 0:
13 |                oLightsDir = numbers
14 |            else:
15 |                oLightsDir = np.vstack( (oLightsDir,np.array(numbers)) )
16 |        return oLightsDir
17 |
18 |
19 | # preizkus funkcije
20 | S = readTxt( os.path.join(DATA, "owl/lights.txt" ))
21 | # preveri, da so dimenzije 12x3
22 | print(S.shape)
```

Izpis v ukazni vrstici je:

```
(12, 3)
```

Vaja 10.2 Ustvarite spremenljivko tipa `list` in zaporedoma od $k=0, \dots, 11$ naložite vse dane barvne slike `owl.k.tif`. Sliki z indeksom k pripada smer svetila v matriki `oLightDir` z indeksom $[k, :]$. V spremenljivko `numpy.array` naložite tudi sliko maske `owl.mask.tif`.

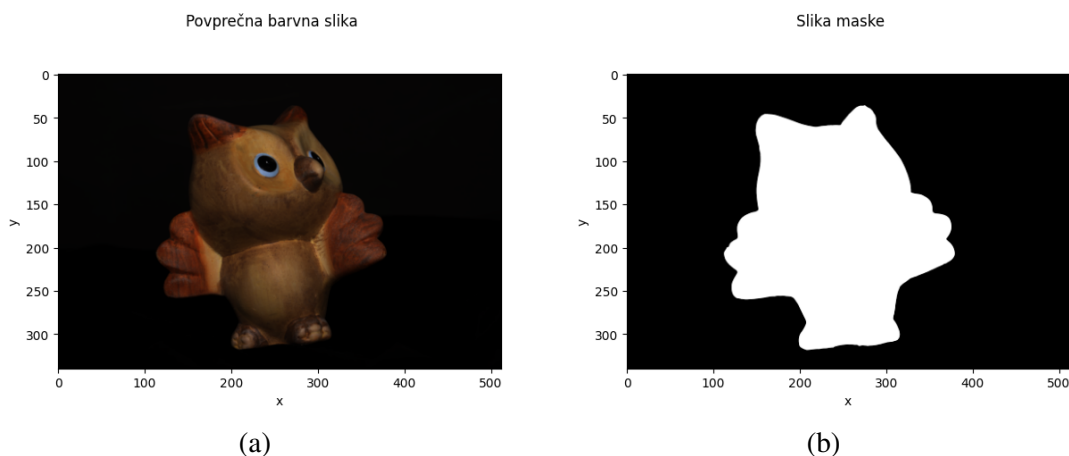
- Izračunajte povprečno barvno sliko, npr. `iMeanImage`.
- Pretvorite vse slike v sivinske slike in jih shranite v spremenljivko tipa `numpy.array`, npr. `iImages` z dimenzijami $Y \times X \times 12$, kjer sta X, Y dimenziji 2D slike.
- Pretvorite sliko maske v sivinsko sliko, npr. `iMask`.

```

1 # naloži masko
2 iMask = rvlb.colorToGray( rvlb.loadImage( "../owl/owl.mask.tif" ) )
3 # naloži barvne slike
4 iImages = []
5 for i in range(12):
6     iImages.append( rvlb.loadImage( f"../owl/owl.{i}.tif" ) )
7 # izračunaj povprečno sliko
8 iMeanImage = np.mean(np.dstack(iImages).reshape(
9     [*iImages[0].shape[:2], 12, 3]).astype('float'), axis=2).astype('uint8')
10 # pretvori barvne v sivinske slike
11 iImagesG = [colorToGray(iImage) for iImage in iImages]
12 # prikaži masko in povprečno barvno sliko
13 rvlb.showImage(iMask, 'Slika maske')
14 rvlb.showImage(iMeanImage, 'Povprečna barvna slika')

```

Rezultat je prikazan na sliki 10.3.



Slika 10.3: Prikaz (a) povprečne barvne slike preko vseh 12 slik in (b) slika maske področja zanimanja.

Vaja 10.3 Napišite funkcijo, ki izračuna normale na površino objekta n iz treh slik objekta:

```

1 def computeNormals( iImages, iMask, iLightDir ):
2     # Tu napišite python kodo funkcije
3     return oNormals

```

kjer je `iImages` matrika dimezij $Y \times X \times 3$ (X, Y sta dimenziji 2D slike), `iMask` je maska

objekta in `iLightDir` matrika 3×3 s smermi osvetlitve, ki ustrezajo slikam v matriki `iImages`. Funkcija vrne 3D matriko nenormaliziranih normal $\tilde{n}$ dimenzij $Y \times X \times 3$ v spremenljivki `oNormals`. Preizkusite delovanje funkcije s poljubnimi tremi vhodnimi slikami ($N = 3$), pri tem pa pazorno izberite vhodne slike glede na pripadajoče smeri osvetlitve s_j .

Prikažite povprečno barvno sliko `iMeanImage` in izrišite normale v obliki vektorskega polja s funkcijo `quiver()` v knjižnici `matplotlib.pyplot`. Preverite ali so smeri normal smiselne glede na dano sliko 3D oblike. ■

Funkcija `computeNormals(iImages, iMask, iLightDir)` izračuna normale površine objekta s fotometrično stereo tehniko na podlagi več slik pod različnimi svetlobnimi pogoji. Najprej preveri število vhodnih svetlobnih smeri (`iLightDir`) in izračuna psevdoinverz matrike smeri S^+ : če so podane natanko tri smeri in je zato matrika smeri kvadratna uporabi navadno inverzijo (S^{-1}), sicer pa psevdoinverz prek metode najmanjših kvadratov ($S^+ = (S^T S)^{-1} S^T$). Nato slike (`iImages`) preoblikuje v 2D matriko (pikslov $\times$ slik) in izračuna normale kot linearno kombinacijo intenzitet pikslov s koeficienti iz S^+ . Piksle zunaj maske (`iMask`) nastavi na nič in normale vrne v obliki 3D matrike (višina $\times$ širina $\times$ 3), kjer zadnja dimenzija predstavlja komponente normal (n_x, n_y, n_z) za vsako točko površine. Rešitev zapišemo kot:

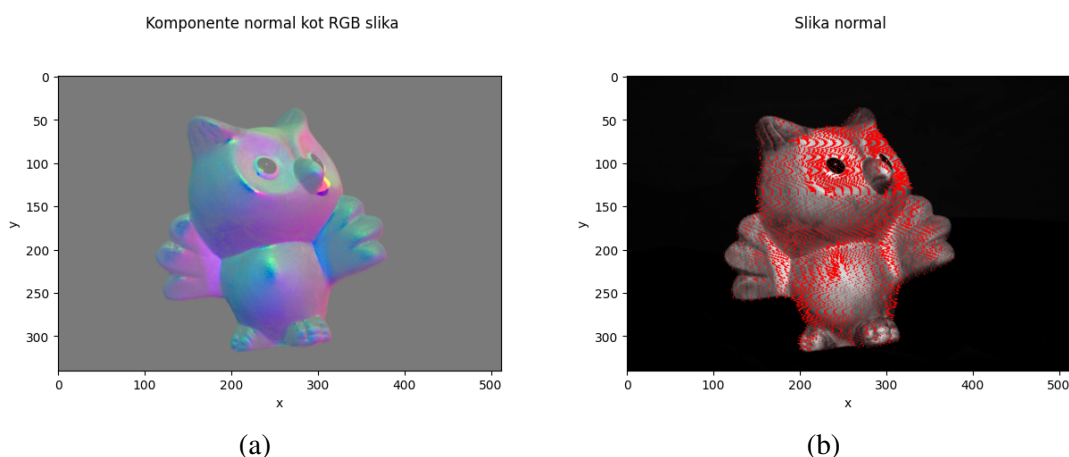
```
1 def computeNormals( iImages, iMask, iLightDir ):
2     """Izracun normal na površino objekta iz treh ali vec slik"""
3     if iLightDir.shape[0] == 3:
4         Sinv = np.linalg.inv( iLightDir.transpose() )
5     else:
6         S = np.dot( iLightDir.transpose(), iLightDir )
7         Sinv = np.dot( np.linalg.inv(S), iLightDir.transpose() )
8         Sinv = Sinv.transpose()
9     shapeImages = iImages.shape
10    iImages = iImages.reshape(
11        (iImages.shape[0]*iImages.shape[1],iImages.shape[2]))
12    oNormals = np.dot( iImages, Sinv )
13    oNormals[iMask.flatten() == 0] = 0
14    oNormals = oNormals.reshape( (shapeImages[0],shapeImages[1],3) )
15    return oNormals
16
17 # preizkus funkcije
18 idx = [1, 5, 9] # izberemo tri vrednosti med 0 in 11
19 # pripravimo vhodne spremenljivke
20 iImagesG_in = np.dstack( [iImagesG[idx[i]] for i in range(len(idx))] )
21 iLightsDir_in = S[idx,:]
22 # izračun normal
23 oNormals = computeNormals( iImagesG_in.astype('float'), iMask, iLightsDir_in )
24 # prikažemo komponente normal kot RGB sliko
25 oNormalsRGB = ( oNormals - np.mean(oNormals) ) / np.std(oNormals)
26 thrZ = 6.0
27 oNormalsRGB[oNormalsRGB>thrZ] = thrZ
28 oNormalsRGB[oNormalsRGB<=-thrZ] = -thrZ
29 oNormalsRGB = 255.0 * ( oNormalsRGB - np.min(oNormalsRGB) ) / \
30     ( np.max(oNormalsRGB) - np.min(oNormalsRGB) )
31 rvlb.showImage(oNormalsRGB.astype('uint8'), 'Komponente normal kot RGB slika')
32 # prikažemo normale kot vektorje
33 my, mx = np.where(iMask>0)
34 plt.imshow(np.mean(iImagesG_in, axis=2))
35 plt.xlabel('x'); plt.ylabel('y')
36 gx, gy = np.meshgrid(range(iImagesG_in.shape[1]), range(iImagesG_in.shape[0]),
```

```

        indexing='xy')
37 plt.quiver( gx[my,mx], gy[my,mx], \
38             oNormals[my,mx,0], \
39             oNormals[my,mx,1], color='r',scale=10000)
40 plt.suptitle('Slika normal')

```

Rezultat je prikazan na sliki 10.4. Glede na intuicijo lahko iz slik razberemo, da ima model sovice privzdignjen trebušček, torej bodo komponente normal v čelni smeri približno nič v središču trebuščka in usmerjene navzven v neposredni okolici.



Slika 10.4: Prikaz (a) komponent vektorja normale kot RGB vrednosti oz. RGB sliko in (b) povprečna sivinska slika in vrisani vektorji normal (prvi dve komponenti).

Vaja 10.4 Napišite funkcijo, ki izračuna albedo oz. odbojnost ρ površine objekta s pomočjo slike nenormaliziranih normal $\tilde{n}$:

```

1 def computeAlbedo( iNormals, iMask ):
2     # Tu napišite python kodo funkcije
3     return oAlbedo

```

kjer je $iNormals$ matrika z dimezijami $Y \times X \times 3$ (X, Y sta dimeziji 2D slike, vsaka normala $\tilde{n}$ pa ima tri komponente), $iMask$ je maska objekta z dimezijami $Y \times X$. Funkcija vrne 2D matriko oz. sliko albeda ρ z dimezijami $Y \times X$ v spremenljivki $oAlbedo$. Zagotovite, da bodo vrednosti v $oAlbedo$ na območju $[0, 1]$.

Preizkusite delovanje funkcije za sliko normal, ki ste jo izračunali pri prejšnji nalogi. ■

Albedo oz. odbojnost določimo kot magnitudo vektorja ocenjenih normal na površino:

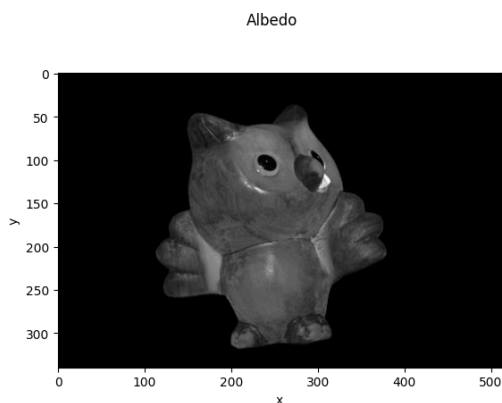
```

1 def computeAlbedo( iNormals, iMask ):
2     """Izračunaj albedo iz nenormaliziranih normal"""
3     oAlbedo = np.sqrt( np.sum(iNormals**2.0, axis=2) )
4     oAlbedo[ iMask == 0 ] = 0
5     oAlbedo = oAlbedo / np.max( oAlbedo )
6     return oAlbedo
7
8 # preizkus funkcije
9 oAlbedo = computeAlbedo( oNormals, iMask )

```

```
10 | rvlb.showImage( oAlbedo, 'Albedo' )
```

Rezultat prikazuje slika 10.5, pri čemer manjše vrednosti albeda sovpadajo s temnimi barvami modela sovice, večje vrednosti pa s svetlimi barvami.



Slika 10.5: Slika parametra albedo oz. odbojnost ρ .

Vaja 10.5 Napišite funkcijo, ki rekonstruira 3D obliko objekta oz. vsaki točki na objektu pripiše globino $z(x,y)$ z analizo odvodov na površino in reševanjem linearnega sistema enačb:

```
1 | def computeDepthLinSys( iNormals, iMask ):
2 |     # Tu napišite python kodo funkcije
3 |     return oDepth
```

kjer je $iNormals$ matrika dimezij $Y \times X \times 3$, $iMask$ je maska objekta z dimezijami $Y \times X$. Poskrbite, da bodo normale $iNormals$ podane v normalizirani obliki n . Pri tej nalogi rešujete linearni sistem enačb $Mz = p$, pri čemer matriko M (dimezije $2XY \times XY$) inicializirate z ukazom `csc_matrix()` v knjižnici `scipy.sparse`, elemente pa v matriko vpisujete posamezno pri ustreznih indeksih. Rešitev sistema enačb dobite s klicem funkcije `lsqr()` v knjižnici `scipy.sparse.linalg`. Vaša funkcija naj vrne 2D matriko z globino $z(x,y)$ dimezij $Y \times X$ v spremenljivki `oDepth`.

Preizkusite delovanje funkcije s sliko `normal`, ki ste jo izračunali pri prvi nalogi. Prikažite rekonstruirano 3D površino s funkcijo `plot_surface()` v knjižnici `mpl_toolkits.mplot3d` in obarvajte površino glede na povprečno barvno sliko `iMeanImage`. ■

Funkcija `computeDepthLinSys(iNormals, iMask)` rekonstruira 3D globino iz normal s fotometrične stereo tehnike z reševanjem linearnega sistema parcialnih diferencialnih enačb. Najprej identificira veljavne piksele z masko (`iMask`) in pripravi redko matriko $M \in \mathbb{R}^{2N \times N}$ (kjer je N število aktivnih točk), ki kodira diskretne aproksimacije parcialnih odvodov $\frac{\partial z}{\partial x} = -\frac{n_x}{n_z}$ in $\frac{\partial z}{\partial y} = -\frac{n_y}{n_z}$ za vsako točko, kjer (n_x, n_y, n_z) predstavljajo komponente normal. Sistem $Mz = v$ se reši z metodo najmanjših kvadratov (`scipy.sparse.linalg.lsqr`), kjer z predstavlja neznane vrednosti globine, nato pa se rezultat normalizira in preslika nazaj v 2D prostor, pri čemer ohrani samo vrednosti znotraj maske. Končna globina se vrne kot 2D polje z enakimi dimezijami kot vhodna maska. Rešitev zapišemo:

```
1 | def computeDepthLinSys( iNormals, iMask ):
2 |     """Izračunaj globino z reševanjem linearnega sistema enačb"""
3 |     from scipy.sparse import csc_matrix
```



```

4     from scipy.sparse import linalg as splinalg
5     # določi velikosti matrik
6     my, mx = np.where(iMask>0)
7     Ncols = my.size
8     Nrows = 2*Ncols
9     # inicializiraj spremenljivke
10    rcData = []
11    addToM = lambda x : rcData.append( x )
12    v = np.zeros( (Nrows,1) )
13    index = np.zeros_like( iMask ).astype('uint32')
14    index[my,mx] = range(mx.size)
15    # sprehodi se preko vseh elementov
16    for i in range(mx.size):
17        nx = iNormals[my[i],mx[i],0]
18        # obrni znak zaradi menjave koordinatnega sistema
19        ny = -iNormals[my[i],mx[i],1]
20        nz = iNormals[my[i],mx[i],2]
21        # parcialni odvod po x
22        if index[my[i],mx[i]+1]>0:
23            addToM( (2*i,i,1) )
24            addToM( (2*i,index[my[i],mx[i]+1],-1) )
25            v[ 2*i,0 ] = nx / nz
26        # parcialni odvod po y
27        if index[my[i]+1,mx[i]]>0:
28            addToM( (2*i+1,i,1) )
29            addToM( (2*i+1,index[my[i]+1,mx[i]],-1) )
30            v[ 2*i+1,0 ] = ny / nz
31    v[np.isnan(v)] = 0
32    # definiraj redko matriko
33    rows = [rcData[i][0] for i in range(len(rcData))]
34    cols = [rcData[i][1] for i in range(len(rcData))]
35    data = [rcData[i][2] for i in range(len(rcData))]
36    M = csc_matrix((data, (rows, cols)), shape=(Nrows, Ncols))
37    # reši sistem enačb
38    b = splinalg.lsqr(M,v.flatten(),show=True)
39    oDepth_t = b[0] - np.min(b[0])
40    # ustvari izhodno polje
41    oDepth = np.zeros_like( iMask ).astype('float')
42    oDepth[my,mx] = oDepth_t
43    return oDepth
44
45    # preizkus funkcije
46    oDepth = computeDepthLinSys( oNormals, iMask )
47    # prikazi kot globinsko sliko
48    rvlib.showImage(oDepth, 'oDepth')

```

V ukazni vrstici se izpišejo iteracije reševanja sistema enačb:

```

LSQR          Least-squares solution of  Ax = b
The matrix A has 95314 rows and 47657 columns
damp = 0.0000000000000000e+00    calc_var =      0
atol = 1.00e-06                  conlim = 1.00e+08
btol = 1.00e-06                  iter_lim =   95314

```

Itn	x[0]	r1norm	r2norm	Compatible	LS	Norm A	Cond A
-----	------	--------	--------	------------	----	--------	--------

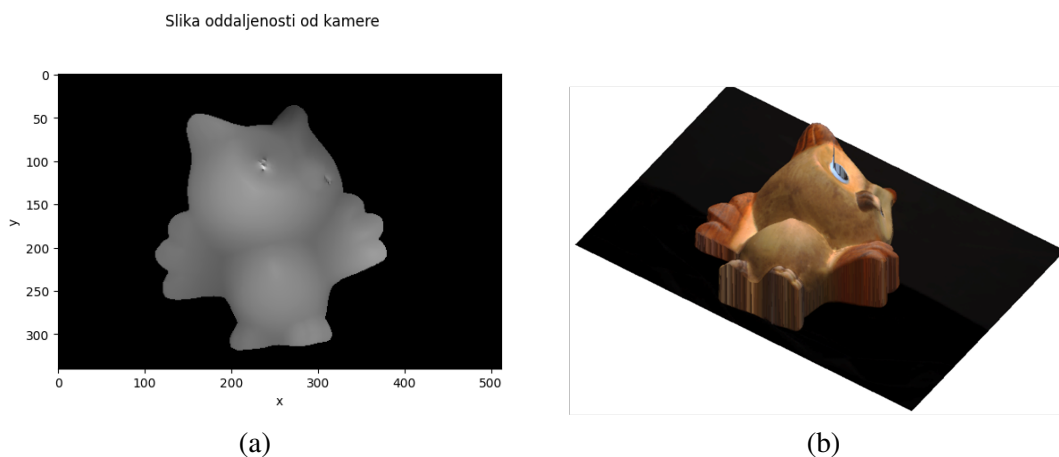
```

0  0.00000e+00  1.288e+03  1.288e+03  1.0e+00  8.2e-04
...
10 -4.60457e+00  1.013e+03  1.013e+03  7.9e-01  1.2e-02  6.4e+00  2.0e
    ↪ +01
...
444 -8.38808e+01  9.861e+02  9.861e+02  7.7e-01  9.2e-06  4.2e+01  4.1e
    ↪ +03
445 -8.38773e+01  9.861e+02  9.861e+02  7.7e-01  8.9e-06  4.2e+01  4.1e
    ↪ +03
...

istop =      2    r1norm = 9.9e+02    anorm = 4.9e+01    arnorm = 4.8e-02
itn    =    604    r2norm = 9.9e+02    acond = 5.6e+03    xnorm  = 4.9e+03

```

Rezultat določanje 3D oblike je slika globine (ang. *depth image*), kot prikazuje slika 10.6a ki jo lahko nadalje pretvorimo v 3D trikotniško mrežo, dodamo barvno teksturo (npr. povprečno barvno sliko `iMeanImage`) in prikažemo v 3D oseh – smiselnost rekonstrukcije enostavno preverimo na tem prikazu (slika 10.6b).



Slika 10.6: Prikaz (a) slike globine in njena (b) upodobitev v obliki 3D površine z originalno barvno teksturo.

10.5 Naloge in vprašanja

Poročilo naloge pripravite v obliki Python skripte, ki naj izvede zahtevane izračune oz. klice funkcij in izriše pripadajoče slike oz. grafe. Vaša dolžnost je, da v skripti uvozite ustrezne Python knjižnice in dodate morebitne podporne skripte. Skripta se mora izvesti in pustvariti rezultate, kot se zahtevajo v nalogah. Na vprašanja odgovorite v obliki komentarja, naprimer `# Naloga X.1: To je moj odgovor..`

Naloga 10.1 Prikažite sliko normal $\tilde{n}$ kot barvno sliko, pri čemer poskrbite za ustrezno pretvorbo vrednosti komponent normal na območje $[0, 255]$.

Naloga 10.2 Popravite funkcijo `computeNormals()` tako, da bo omogočala izračun normal na površino objekta n iz poljubnega števila slik objekta N . Vhodni parameter `iImages` naj bo matrika dimezij $Y \times X \times N$ (X, Y sta dimenziji 2D slike, N pa število slik z različno osvetlitvijo), `iMask` binarna maska objekta z dimenzijami $Y \times X$ in `iLightDir` matrika $N \times 3$ s smermi osvetlitve, ki ustrezajo slikam v matriki `iImages`. Funkcija vrne 3D matriko nenormaliziranih normal $\tilde{n}$ dimenzij

$Y \times X \times 3$ v spremenljivki `oNormals`. Preizkusite delovanje funkcije z vsemi danimi vhodnimi slikami ($N = 12$) in pripadajočimi smermi osvetlitve s_j .

Prikažite povprečno barvno sliko `iMeanImage` in izrišite normale v obliki vektorskega polja s funkcijo `quiver()`. Na katerem delu objekta se smeri normal izboljšajo glede na smeri normal, ki ste jih določili le s tremi slikami? Obrazložite odgovor.

Naloga 10.3 Razširite funkcijo `computeNormals()` tako, da bo omogočala robusten izračun normal na površino objekta n z uteževanjem s sivinsko vrednostjo slik, tj. osnovna enačba za zapis linearne sistema ima obliko $I_j^2 = I_j (\rho s_j^T \cdot n)$. Preizkusite delovanje funkcije z vsemi danimi vhodnimi slikami ($N = 12$) in pripadajočimi smermi osvetlitve s_j .

Prikažite povprečno barvno sliko `iMeanImage` in izrišite normale v obliki vektorskega polja s funkcijo `quiver()`. Na katerem delu objekta se smeri normal izboljšajo glede na smeri normal, ki ste jih določili le s tremi slikami? Obrazložite odgovor.

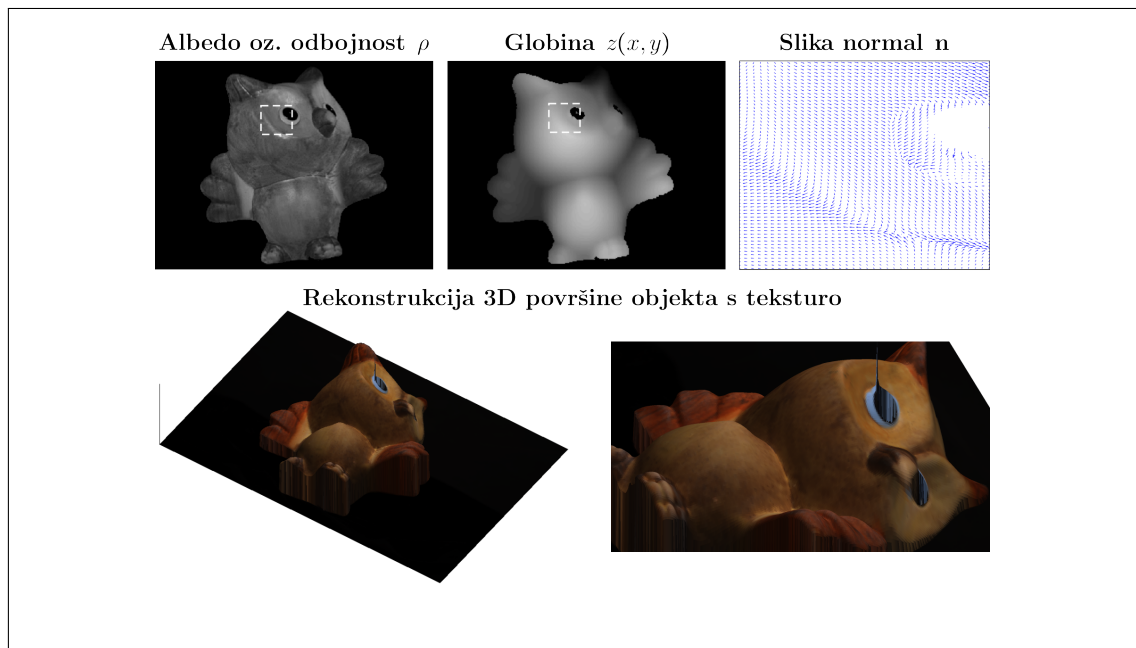
Naloga 10.4 Uporabite posodobljeno funkcijo `computeNormals()` za rekonstrukcijo 3D površine objekta z vsemi vhodnimi slikami ($N = 12$) in določite sliko globine $z(x, y)$ s funkcijo `computeDepthLinSys()`. Prikažite rekonstruirano 3D površino s funkcijo `in` jo obarvajte glede na povprečno barvno sliko `iMeanImage`.

Naloga 10.5 Napišite funkcijo, ki rekonstruira 3D obliko objekta oz. vsaki točki na objektu pripiše globino $z(x, y)$ z integriranjem normal n po poljubni krivulji:

```
1 def computeDepthIntegrate( iNormals, iMask ):
2     # Tu napišite python kodo funkcije
3     return oDepth
```

kjer je `iNormals` matrika dimezij $X \times Y \times 3$, `iMask` je maska objekta z dimezijami $X \times Y$. Poskrbite, da bodo normale `iNormals` podane v normalizirani obliki n . Funkcija vrne 2D matriko z globino $z(x, y)$ dimezij $X \times Y$ v spremenljivki `oDepth`. Preizkusite delovanje funkcije s sliko normal, ki ste jo izračunali pri prvi nalogi. Prikažite rekonstruirano 3D površino s funkcijo `surf()`.

Rezultati nekaterih nalog:



11. Sledenje in analiza gibanja

Na podlagi zaporedja slik zajetih z eno ali več kamerami lahko analiziramo gibanje objektov v 3D prostoru in s to informacijo načrtujemo povratno-zančno vodenje, manipulacijo, sledenje ter izvajamo rekonstrukcijo in detekcijo teh objektov. Postopek analize gibanja običajno sestavlja več osnovnih postopkov, npr. postopek za robustno zaznavanje objektov zanimanja, iskanje korespondenc med zaporedno zajetimi slikami in poravnava slik. Pri vaji boste načrtali postopek sledenja objekta zanimanja oz. *tarče* z uporabo **Lucas-Kanade poravnave slik**.

11.1 Parametrizacija preslikave

Pri tem postopku za sledenje uporabljamo parametrični model preslikave $\mathcal{T} : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ med zaporedno zajetima 2D slikama $I_k(x)$ in $I_{k-1}(x)$ ($x = [x, y]^T$), kjer je prva slika zajeta ob času $t(k)$, druga pa ob $t(k-1)$ in pri čemer je $t(k) > t(k-1)$. Parametrični model $\mathcal{T}(x; p)$ je npr. translacija, podobnostna, afina ali projektivna preslikava (p ima 2, 3, 6 oz. 8 parametrov). Privzamemo, da se med dvema zaporednima slikama I_1 in I_2 ohranja svetlost slikovnih elementov $I_1(x) = I_2(\mathcal{T}(x; p)) + n(x)$ ¹ in za mero podobnosti uporabimo vsoto kvadratov razlik med slikama.

11.2 Linearizacija nelinearne kriterijske funkcije

Optimalne parametre p^* tako dobimo z minimizacijo izraza

$$p^* = \arg \min_p \sum_x [I_2(\mathcal{T}(x; p)) - I_1(x)]^2.$$

Ker pričakujemo majhne premike tarče med slikama I_1 in I_2 ($p \rightarrow 0$) lahko iščemo le optimalen korak Δp^* v parametrih preslikave

$$\Delta p^* = \arg \min_{\Delta p} \sum_x [I_2(\mathcal{T}(x; p + \Delta p)) - I_1(x)]^2,$$

¹ $n(x) \sim \mathcal{N}(0, \sigma^2)$ je Gaussov šum.

parametre preslikave pa dobimo kot $p \leftarrow p + \Delta p^*$. Optimalni korak Δp^* izračunamo brez optimizacije tako, da izraz $I_2(\mathcal{T}(x; p + \Delta p))$ aproksimiramo s Taylorjevo vrsto kot

$$I_2(\mathcal{T}(x; p + \Delta p)) \approx I_2(\mathcal{T}(x; p)) + \nabla I_2(\mathcal{T}(x; p)) \frac{\partial \mathcal{T}}{\partial p} \Delta p,$$

kjer je $\nabla I_2(\mathcal{T}(x; p)) = [I_x, I_y]$ gradient slike I_2^2 pri trenutni preslikavi $\mathcal{T}(x; p)$. Izraz $\partial \mathcal{T} / \partial p$ predstavlja Jacobijevo matriko prvih odvodov in zavisi od izbrane parametrične preslikave $\mathcal{T}$. Razliko med slikama v prvi enačbi lahko zapišemo v matrični obliki kot $I_t + B\Delta p$, kjer je $I_t = I_2(\mathcal{T}(p)) - I_1$; I_k je $N \times 1$ vektor sivinskih vrednosti slike. Izrazi za B pri translaciji in afini preslikavi so podani spodaj.

Translacija:

$$\mathcal{T}(x; p) = \begin{bmatrix} x + t_x \\ y + t_y \end{bmatrix}$$

$$\frac{\partial \mathcal{T}}{\partial p} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\nabla I_2 \frac{\partial \mathcal{T}}{\partial p} = [I_x, I_y]$$

$$B = [I_x, I_y] \in \mathbb{R}^{N \times 2}$$

Afina preslikava:

$$\mathcal{T}(x; p) = \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\frac{\partial \mathcal{T}}{\partial p} = \begin{bmatrix} x & 0 & y & 0 & 1 & 0 \\ 0 & x & 0 & y & 0 & 1 \end{bmatrix}$$

$$\nabla I_2 \frac{\partial \mathcal{T}}{\partial p} = [I_x x, I_y x, I_x y, I_y y, I_x, I_y]$$

$$B = [I_x X, I_y X, I_x Y, I_y Y, I_x, I_y] \in \mathbb{R}^{N \times 6}$$

11.3 Zaprto-zančna posodobitev preslikave

Optimalen korak Δp^* v parametrih preslikave se nato izraža kot

$$\Delta p^* = \arg \min_{\Delta p} (I_t + B\Delta p)^T (I_t + B\Delta p),$$

njegovo vrednost pa dobimo tako, da odvajamo gornjo enačbo na Δp in odvod izenačimo z 0. Dobimo linearen sistem enačb za Δp^* , ki ima rešitev

$$\Delta p^* = -(B^T B)^{-1} B^T I_t.$$

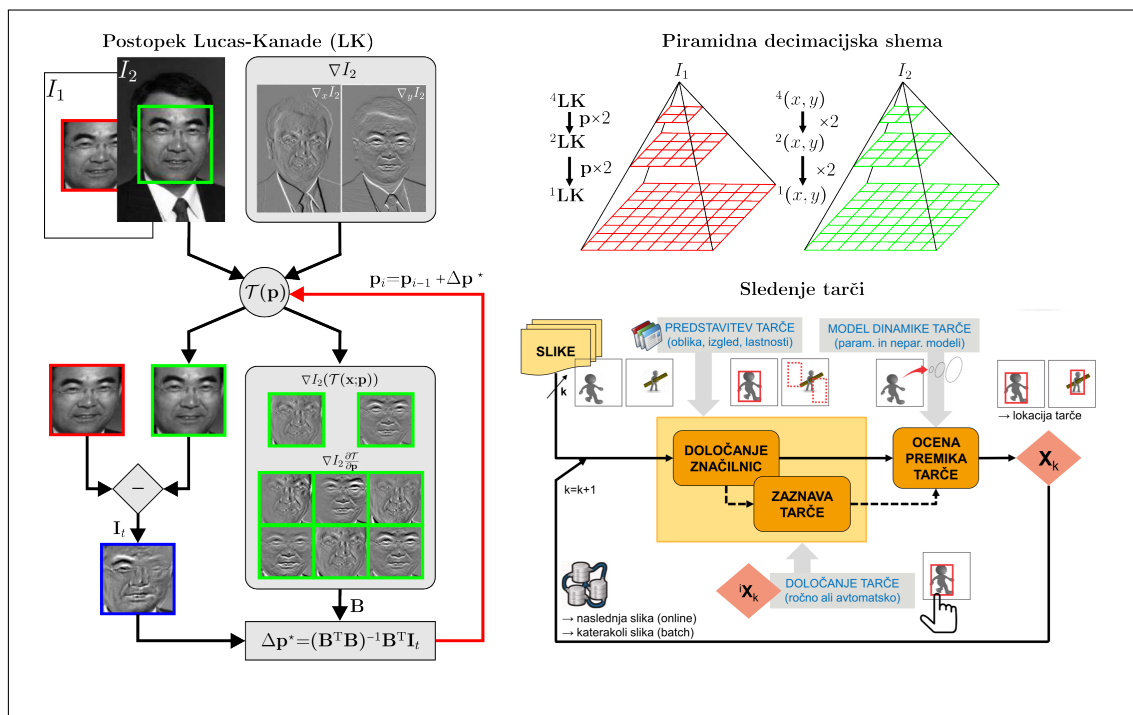
Postopek Lucas-Kanade izvajamo iterativno tako, da zaporedoma izračunavamo optimalen korak Δp^* in posodabljam parametre preslikave $p_i = p_{i-1} + \Delta p^*$ do nekega maksimalnega števila iteracij $i_{\max}$ ali dokler $\|p_i - p_{i-1}\| < \varepsilon$. Potek postopka je prikazan na spodnji sliki.

11.4 Piramidna implementacija

Postopek Lucas-Kanade konvergira v najbližji lokalni minimum, zato je potrebno za pri večjih premikih tarče med slikami uporabiti dober začetni približek parametrov p . Dober začetni približek lahko dobimo z uporabo piramidne decimacije tako, da poravnamo slike najprej pri redkejšem vzorčenju, parametre pa nato izboljšamo pri gostejšem vzorčenju slike. Gostoto vzorčenja običajno spreminjamo s korakom $\times 2$, glede na spremembo koraka vzorčenja pa enako spreminjamo tudi parametre p .

Sledenje tarči v videu izvedemo tako, da v prvi sliki ročno ali avtomatsko določimo središče tarče (x_c, y_c) in velikost pravokotnega okna (w, h) tako, da okno vsebuje celotno tarčo. Če poznamo središče tarče v sliki I_1 , potem z Lucas-Kanade postopkom poravnamo sliko I_1 na naslednjo zajeto sliko I_2 in glede na dobljeno preslikavo $\mathcal{T}(x; p)$ posodobimo središče tarče v sliki I_2 . Postopek ponavljamo za vse naslednje zaporedne pare zajetih slik $\{I_k, I_{k+1}\}, k = 1, \dots, T - 1$.

$${}^2\nabla I_2 = \begin{bmatrix} \frac{\partial I_2}{\partial x} & \frac{\partial I_2}{\partial y} \end{bmatrix} = [I_x, I_y]$$



11.5 Vaje z rešitvami

Gradivo za vajo vsebuje videa `video1.avi` in `video2.avi` zajeta z nadzorno kamero s pogledom na pločnik iz ptičje perspektive. V Python skripti `funkcije.py` so pripravljene funkcije, ki jih boste potrebovali za izvedbo vaje, pri kateri boste načrtali postopek za sledenje gibanja oseb, ki se sprehajajo po pločniku. Koda vaje je dostopna na [Git repozitoriju](#).

Poleg gradiva boste za nalaganje videov v Python boste potrebovali knjižnico `ffmpeg`, ki jo najdete na spletnem naslovu <https://www.ffmpeg.org/download.html>.

V Windows okolju si program `ffmpeg.exe` skopirajte v vašo delovno mapo. V MacOS okolju v ukazni vrstici izvedite ukaz:

```
brew install ffmpeg
```

V Linux okolju, na primer Ubuntu, v ukazni vrstici izvedite:

```
sudo apt update
sudo apt install ffmpeg
```

V ukazni vrstici (*vsa okolja*) pojdite v delovno mapo vaje in preverite delovanje tako, da izvedete:

```
ffmpeg -version
```

Odziv v primeru uspešne namestitve bo približno takole:

```
ffmpeg version 7.1.1 Copyright (c) 2000-2025 the FFmpeg developers
...
```

Nato v Python uvozimo knjižnice:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import matplotlib.animation as animation
4
5 from rvlb import loadVideo, showVideo, showImage, transformImage, \
```

```
6 drawPathToFrame, imageGradient, transAffine2D, decimateImage2D, \
7 interpolate1Image2D
```

Vaja 11.1 Okvirje v video datoteki `video1.avi` naložite v Python spremenljivko `numpy.array` s pomočjo funkcije `loadVideo()` in prikažite s funkcijo `showVideo()` kot prikazuje naslednja skripta:

```
1 # naloži video
2 oVideo = loadVideo( 'video1.avi' )
3 # prikaži video
4 showVideo( oVideo )
5 # velikost polja videa
6 print('\nVelikost polja videa:\n', oVideo.shape)
```

V ukazni vrstici se izpisujejo podatki o branju videa:

```
Berem okvir 1 ...
frame= 50 fps=0.0 q=-0.0 size= 59535KiB time=00:00:02.32 bitrate=210220.1
  ↳ kbits/s speed=4.61x
...
Berem okvir 178 ... koncano!
[out#0/image2pipe @ 0x159732d70] video:215055KiB audio:0KiB subtitle:0KiB other
  ↳ streams:0KiB global headers:0KiB muxing overhead: 0.000000%
frame= 177 fps= 64 q=-0.0 Lsize= 215055KiB time=00:00:07.08 bitrate=248832.0
  ↳ kbits/s speed=2.56x

Velikost polja videa:
(576, 720, 177)
```

Iz izpisa je razvidno, da ima slika v videu velikost 576×720 slikovnih elementov in da je sestavljena iz 177 okvirjev. Z ukazom `ffmpeg -i video1.avi` lahko ugotovimo, da se video predvaja s 25 okvirji na sekundo (fps; ang. *frames per second*). Izris okvirja videa v prikaznem oknu je na sliki 11.1.



Slika 11.1: Okvir videa `video1.avi` pretvorjen v sivinsko sliko.

Vaja 11.2 Napišite funkcijo za poravnavo dveh sivinskih slik `iImgFix` in `iImgMov` z Lucas-Kanade postopkom:

```
1 def regLucasKanade( iImgFix, iImgMov, iMaxIter ):
2     # Tu napišite python kodo funkcije
3     return oPar, oImgReg
```

kjer je `iImgMov` premična slika, ki jo želimo poravnati na fiksno sliko `iImgFix`. Parameter `iMaxIter` predstavlja število korakov Lucas-Kanade postopka. Funkcija vrne dvovrstični vektor parametrov `oPar` s translacijama $p = [t_x, t_y]^T$ in preslikano premično sliko v spremenljivki `oImgReg`. Vhodni slik `iImgFix` in `iImgMov` in izhodna slika `oImgReg` imajo enake dimenzije.

Preizkusite delovanje poravnave na prvi sliki v videu `video1.avi` tako, da sliko preslikate s parametri $p = [-5, 3]^T$ in nato poravnate z Lucas-Kanade postopkom. Ocenjeni parametri preslikave `oPar` morajo biti približno enaki dejanskim parametrom p . ■

Funkcija `regLucasKanade()` implementira iterativni Lucas-Kanade algoritem za poravnavo slik, ki minimizira razliko med fiksno oz. referenčno sliko (`iImgFix`) in preslikano premično sliko (`iImgMov`) z optimizacijo affine transformacije. Algoritem najprej izračuna gradienta G_x in G_y gibljive slike, nato v vsaki iteraciji: (1) preslika gibljivo sliko in njene gradiente z aktualnimi parametri $\mathbf{p} = (t_x, t_y)$, (2) izračuna razliko slik I_t , (3) sestavi Jacobijevo matriko $\mathbf{B}$ iz gradientov, (4) reši linearni sistem $\mathbf{B}^T \mathbf{B} \Delta \mathbf{p} = \mathbf{B}^T I_t$ za posodobitev parametrov $\mathbf{p} \leftarrow \mathbf{p} + \Delta \mathbf{p}$ z metodo najmanjših kvadratov. Postopek se ponavlja do končnega števila iteracij (`iMaxIter`), pri čemer se na koncu vrnejo optimizirani parametri in poravnana slika. Rešitev lahko zapišemo:

```
1 def regLucasKanade(iImgFix, iImgMov, iMaxIter, oPar = (0,0), iVerbose=True):
2     """Postopek poravnave Lucas-Kanade"""
3     # pretvori vhodne slike v numpy polja tipa float
4     iImgType = np.asarray(iImgMov).dtype
5     iImgFix = np.array(iImgFix, dtype='float')
6     iImgMov = np.array(iImgMov, dtype='float')
7     # določi začetne parametre preslikave
8     oPar = np.array(oPar)
9     # izračunaj prva odvoda slike
10    Gx, Gy = imageGradient(iImgMov)
11    # v zanki iterativno posodablja parametre
12    for i in range(iMaxIter):
13        # določi preslikavo pri trenutnih parametrih
14        oMat2D = transAffine2D(iTrans=oPar)
15        # preslikaj premično sliko in slike odvodov
16        iImgMov_t = transformImage(iImgMov, oMat2D)
17        Gx_t = transformImage(Gx, oMat2D)
18        Gy_t = transformImage(Gy, oMat2D)
19        # izračunaj sliko razlike in sistemsko matriko
20        I_t = iImgMov_t - iImgFix
21        B = np.vstack((Gx_t.flatten(), Gy_t.flatten())).transpose()
22        # reši sistem enačb
23        invBtB = np.linalg.inv(np.dot(B.transpose(), B))
24        dp = np.dot(np.dot(invBtB, B.transpose()), I_t.flatten())
25        # posodobi parametre
26        oPar = oPar + dp.flatten()
27        if iVerbose: print('iter: %d' % i, ', oPar: ', oPar)
28    # določi preslikavo pri končnih parametrih
29    oMat2D = transAffine2D(iTrans=oPar)
30    # preslikaj premično sliko
```

```

31     oImgReg = transformImage(iImgMov, oMat2D).astype(iImgType)
32     # vrni rezultat
33     return oPar, oImgReg
34
35 # preizkus funkcije
36 oPar = [0, 1] # doloci parametre sintetične affine preslikave
37 iImgFix = oVideo[:, :, 0]
38 iImgMov = transformImage(iImgFix, transAffine2D(iTrans = oPar)) # preslikava
39 # klici Lucas-Kanade poravnavo slik
40 oPar, oImgReg = regLucasKanade(iImgFix, iImgMov, 20)
41 print('parameters: ', oPar)
42 # prikaz slike razlik pred in po poravnavi
43 showImage(iImgFix.astype('float') - iImgMov.astype('float'), 'Razlika pred
    poravnavo')
44 showImage(iImgFix.astype('float') - oImgReg.astype('float'), 'Razlika po
    poravnavi')

```

V preizkusu funkcije smo ustvarili afino preslikavo s translacijo $(t_x, t_y) = (0, 1)$ in jo uporabili za preslikavo prvega okvirja slike. Preslikana slika je *premična*, originalna slika pa *fiksna* za namen preizkušanja funkcije. Funkcija v ukazni vrstici izpiše:

```

iter: 0 , oPar: [ 0.000607 -0.1348281]
iter: 1 , oPar: [ 0.00130562 -0.2682225 ]
iter: 2 , oPar: [ 0.00194521 -0.3935767 ]
iter: 3 , oPar: [ 0.00244312 -0.50428944]
iter: 4 , oPar: [ 0.00275654 -0.59677659]
iter: 5 , oPar: [ 0.00289502 -0.67136986]
iter: 6 , oPar: [ 0.00289771 -0.73070864]
...
iter: 17 , oPar: [ 0.00110536 -0.95931672]
iter: 18 , oPar: [ 0.00098431 -0.9652549 ]
iter: 19 , oPar: [ 8.75018467e-04 -9.70296550e-01]
parameters: [ 8.75018467e-04 -9.70296550e-01]

```

Ocenjeni parametri preslikave po 20 iteracijah so $[0,000875, -0.97029]$, kar dobro kompenzira prej ustvarjeni premik $(t_x, t_y) = (0, 1)$. V sliki 11.2 je prikazana slika razlik sivinskih vrednosti med fiksno sliko ter premično pred in po Lukas-Kanade poravnavi, kjer opazimo da poravnava kompenzira razlike v sivinskih vrednostih na celotnem delu slike.

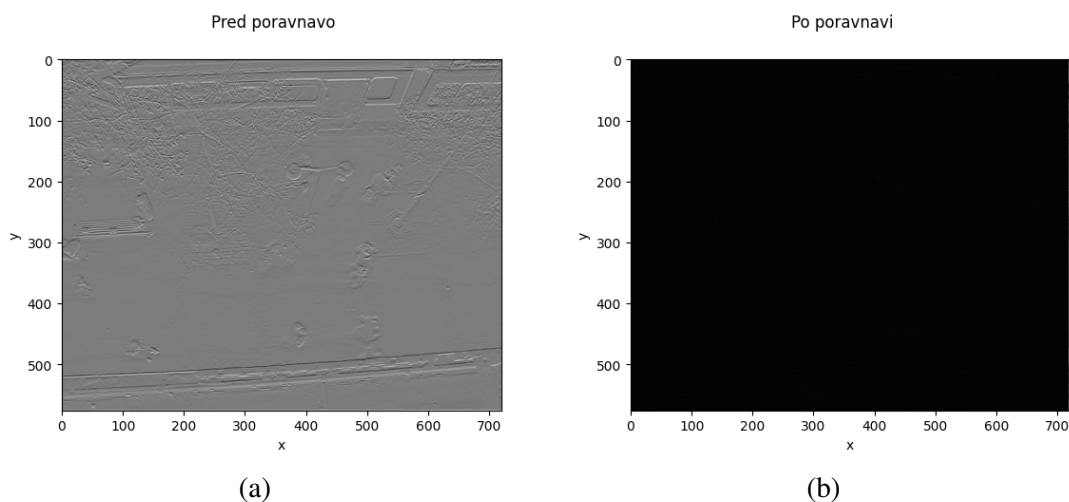
Vaja 11.3 Napišite funkcijo za poravnavo dveh sivinskih slik *iImgFix* in *iImgMov* s piramidnim Lucas-Kanade postopkom:

```

1 def regPyramidLK( iImgFix, iImgMov, iMaxIter, iNumScales ):
2     # Tu napišite python kodo funkcije
3     return oPar, oImgReg

```

kjer je *iImgMov* premična slika, ki jo želimo poravnati na fiksno sliko *iImgFix*. Vhodni parameter *iNumScales* (*s*) je število zaporednih decimacij vhodnih slik *iImgMov* in *iImgFix* s korakom 1:2, ki sestavljajo piramido slik s progresivno manjšim vzorčenjem. Lucas-Kanade postopek izvedemo zaporedno na slikah od najmanjšega do največjega vzorčenja, hkrati pa pri prehodu iz enega na drugi nivo posodobimo začetne parametre kot $p^s = 2 \times p^{s-1}$. Parameter *iMaxIter* predstavlja število korakov Lucas-Kanade postopka v vsakem nivoju piramide.



Slika 11.2: Slika razlik sivinskih vrednosti med fiksno sliko ter premično (a) pred in (b) po Lukas-Kanade poravnavi.

Preizkusite delovanje poravnave na prvi sliki v videu `video1.avi` tako, da sliko preslikate s parametri $p = [50, -30]^T$ in nato poravnate s piramidnim Lucas-Kanade postopkom. Ocenjeni parametri preslikave `oPar` morajo biti približno enaki dejanskim parametrom p . ■

Funkcija `regPyramidLK()` implementira piramidno različico Lucas-Kanade algoritma, ki izvaja poravnavo slik na več nivojih Gaussove piramide. Najprej zgradi piramido slik z uporabo funkcije `decimateImage2D`, kjer se na vsakem nivoju ločljivost zmanjša za faktor 2. Poravnava se začne na najnižji ločljivosti (najmanjša slika v piramidi), kjer se inicializirajo parametri transformacije $\mathbf{p} = (0, 0)$, nato pa se postopek rekurzivno izvaja proti višjim ločljivostim, pri čemer se parametri transformacije na vsakem nivoju skalirajo z faktorjem 2 in natančneje prilagodijo z osnovnim Lucas-Kanade algoritmom (`regLucasKanade`). Ta pristop omogoča robustno poravnavo kljub velikim premikom in izboljšano konvergenco algoritma, saj se najprej rešuje groba poravnava na nizkih ločljivostih, nato pa le natančne prilagoditve na visokih ločljivostih. Rešitev lahko zapišemo:

```

1 def regPyramidLK(iImgFix, iImgMov, iMaxIter, iNumScales, iVerbose=True):
2     """Piramidna implementacija poravnave Lucas-Kanade"""
3     # pretvori vhodne slike v numpy polja tipa float
4     iImgFix = np.array(iImgFix, dtype='float')
5     iImgMov = np.array(iImgMov, dtype='float')
6     # pripravi piramido slik
7     iPyramid = [ (iImgFix, iImgMov) ]
8     for i in range(1, iNumScales):
9         # decimiraj fiksno in premicno sliko za faktor 2
10        iImgFix_2 = decimateImage2D(iImgFix, i)
11        iImgMov_2 = decimateImage2D(iImgMov, i)
12        # dodaj v seznam
13        iPyramid.append((iImgFix_2, iImgMov_2))
14    # določi začetne parametre preslikave
15    oPar = np.array((0, 0))
16    # izvede poravnavo od najmanjše do največje ločljivosti slik
17    for i in range(len(iPyramid)-1, -1, -1):
18        if iVerbose:
19            print('PORAVNAVA Z DECIMACIJO x{}'.format(2**i))
20        # posodobi parametre preslikave na višjo ločljivost

```

```

21     oPar = oPar * 2.0
22     # izvedi poravnavo pri trenutni ločljivosti
23     oPar, oImgReg = regLucasKanade(iPyramid[i][0], iPyramid[i][1], \
24                                   iMaxIter, oPar, iVerbose=iVerbose)
25     # vrne končne parametre in poravnano sliko
26     return oPar, oImgReg
27
28 # preizkus funkcije
29 oPar = [0, 1] # doloci parametre sintetične afine preslikave
30 iImgFix = oVideo[:, :, 0]
31 iImgMov = transformImage(iImgFix, transAffine2D(iTrans = oPar))
32 # kliči piramidno Lucas-Kanade poravnavo slik
33 oPar, oImgReg = regPyramidLK(iImgFix, iImgMov, 20, 3)
34 print('parameters: ', oPar)
35 # prikaz slike razlik pred in po poravnavi
36 showImage(iImgFix.astype('float') - iImgMov.astype('float'), 'Razlika pred
    poravnavo')
37 showImage(iImgFix.astype('float') - oImgReg.astype('float'), 'Razlika po
    poravnavi')

```

Podobno kot pri vaji 11.2 smo v preizkusu funkcije smo ustvarili afino preslikavo s translacijo $(t_x, t_y) = (0, 10)$ in jo uporabili za preslikavo prvega okvirja slike, nato pa jo poravnali na originalno sliko. Funkcija v ukazni vrstici izpiše:

```

PORAVNAVA Z DECIMACIJO x4
iter: 0 , oPar: [-0.00920315 -0.09918953]
iter: 1 , oPar: [-0.01683559 -0.21310131]
...
iter: 19 , oPar: [-0.00567002 -2.3217941 ]
PORAVNAVA Z DECIMACIJO x2
iter: 0 , oPar: [-0.00909037 -4.70598619]
iter: 1 , oPar: [-0.0072928 -4.75628114]
...

iter: 19 , oPar: [ 1.80525073e-05 -9.99871471e+00]
parameters: [ 1.80525073e-05 -9.99871471e+00]

```

Ocenjeni parametri preslikave po 3×20 iteracijah so $[0, 000081, -9.99871]$, kar dobro kompenzira prej ustvarjeni premik $(t_x, t_y) = (0, 10)$. V sliki 11.3 je prikazana slika razlik sivinskih vrednosti med fiksno sliko ter premično pred in po Lukas-Kanade poravnavi. Poravnava kompenzira razlike v sivinskih vrednostih na večini dela slike, le gornji rob kaže razlike, in sicer zaradi način priprave premične slike s preslikovanjem iz domene originalne slike, kjer je sivinska vrednost privzeto enaka 0.

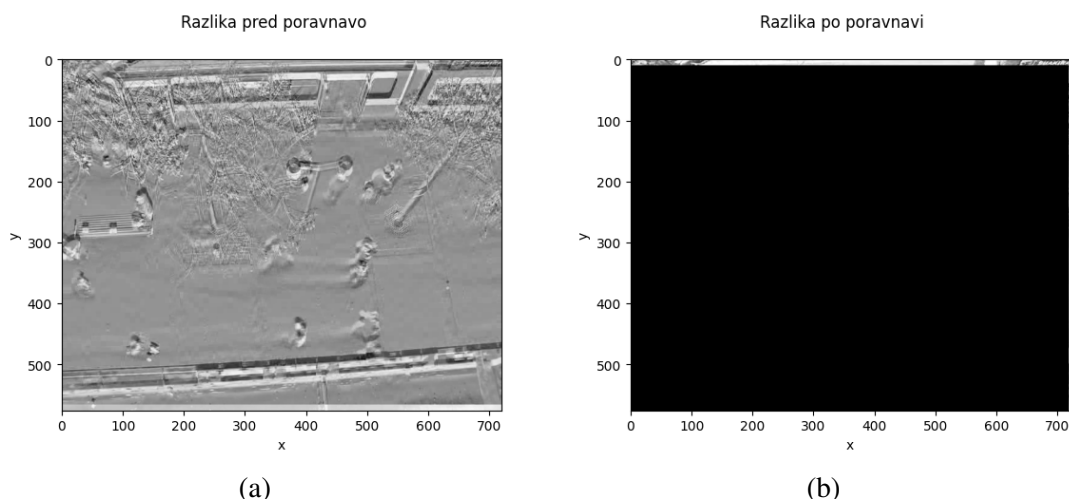
Vaja 11.4 Napišite funkcijo za sledenje izbrani tarči v videu s piramidnim Lucas-Kanade postopkom:

```

1 def trackTargetLK( iVideoMat, iCenterXY, iFrameXY ):
2     # Tu napišite python kodo funkcije
3     return oPathXY

```

kjer je *iVideoMat* video v matriki z dimenzijami $X \times Y \times T$; X, Y sta prostorski, T pa časovna os. Vhodna parametra *iCenterXY* in *iFrameXY* sta dvovrstična vektorja, ki ustrezata središču



Slika 11.3: Slika razlik sivinskih vrednosti med fiksno sliko ter premično (a) pred in (b) po piramidni Lukas-Kanade poravnavi.

tarče (x_c, y_c) in velikost pravokotnega okna (w, h) tarče. Funkcija vrne sled tarče v matriki `oPathXY` dimenzij $2 \times T$, tj. središče tarče v vsakem okvirju videa.

Preizkusite postopek za sledenje tarče na videu `video1.avi`. V prvem okvirju videa ima tarča središče $(x_c, y_c) = [34, 371]^T$ in se nahaja v pravokotnem oknu $(w, h) = [40, 40]^T$. Prikažite video in vrišite sled tarče z uporabo funkcije `drawPathToFrame()` v knjižnici `rvlib` tako, kot je prikazano na sliki 11.4.

Funkcija `trackTargetLK()` implementira sledenje tarče v video zaporedju z uporabo piramidnega Lucas-Kanade algoritma. Algoritem glede na vhodne vrednosti inicializira začetno pozicijo tarče `iCenterXY` in območje interesa velikosti `iFrameXY`, nato za vsak par zaporednih okvirjev v video zaporedju: (1) vzorci območje tarče v trenutnem in naslednjem okvirju z bilinearno interpolacijo (`interpolateImage2D`), (2) izvede piramidno poravnavo (`regPyramidLK`) z maksimalno 30 iteracijami na 3 nivojih piramide za določitev premika $\Delta \mathbf{p} = (\Delta x, \Delta y)$, (3) posodobi pot tarče $\mathbf{oPathXY} \in \mathbb{R}^{N \times 2}$ z dodajanjem novih koordinat.

```

1 def trackTargetLK(iVideoMat, iCenterXY, iFrameXY, iVerbose=True):
2     """Postopek sledenja Lucas-Kanade"""
3     # pretvori vhodni video v numpy polje
4     iVideoMat = np.asarray(iVideoMat)
5     iCenterXY = np.array(iCenterXY)
6     # definiraj izhodno spremenljivko
7     oPathXY = np.array(iCenterXY.flatten()).reshape((1,2))
8     # definiraj koordinate v tarci
9     gx, gy = np.meshgrid(range(iFrameXY[0]), range(iFrameXY[1]))
10    gx = gx - float(iFrameXY[0]-1)/2.0
11    gy = gy - float(iFrameXY[1]-1)/2.0
12    # zazeni LK preko vseh zaporednih okvirjev
13    for i in range(1, iVideoMat.shape[-1]):
14        # vzorčni tarco v dveh zaporednih okvirjih
15        iImgFix = interpolateImage2D(iVideoMat[... , i-1], \
16                                    gx+oPathXY[-1,0], gy+oPathXY[-1,1])
17        iImgMov = interpolateImage2D(iVideoMat[... , i], \
18                                    gx+oPathXY[-1,0], gy+oPathXY[-1,1])
19        # zaženi piramidno LK poravnavo
20        oPar, oImgReg = regPyramidLK(iImgFix, iImgMov, 30, 3, iVerbose=False)

```

```

21     # shrani koordinate
22     oPathXY = np.vstack((oPathXY, oPathXY[-1,:] + oPar.flatten()))
23     print('koordinata tarce: ', oPathXY[-1,:])
24     # vrni spremenljivko
25     return oPathXY
26
27 # preizkus funkcije
28 oPathXY = trackTargetLK(oVideo[...,:], (33,370), (40,40))
29 # prikazi tarčo in pot do tarče v različnih okvirjih videa
30 drawPathToFrame(oVideo, oPathXY, iFrame=1, iFrameSize=(40,40))
31 drawPathToFrame(oVideo, oPathXY, iFrame=100, iFrameSize=(40,40))
32 drawPathToFrame(oVideo, oPathXY, iFrame=170, iFrameSize=(40,40))

```

Končni rezultat funkcije `trackTargetLK()` je matrika poti tarče skozi vse okvirje video zaporedja, ki omogoča vizualizacijo gibanja tarče z uporabo pomožne funkcije `drawPathToFrame`. Rezultat prikazuje slika 11.4. Tarča se pretežno premika od *levega do desnega roba slike*, kar je razvidno tudi iz izpisanih koordinat točk trajektorije tarče, ki jih določi funkcija `trackTargetLK()`:

```

[ 36.55881156 370.14133975]
[ 44.07195853 370.7327816 ]
[ 52.52228952 371.23034241]
[ 61.19684308 371.21558487]
[ 69.80159895 371.20814793]
[ 77.24933027 371.53277712]
[ 84.28658295 372.10005505]
[ 92.32973791 373.3278628 ]
[100.67993777 374.94910248]
[109.45608417 377.45841741]
[117.53670261 379.57140207]
[124.75423269 380.55816688]
[131.91940812 381.01102918]
...
[707.31132931 387.65142739]
[708.62080031 388.2718046 ]

```

Analiza gibanja

Na osnovi analize teh koordinat lahko določimo tudi dolžino poti tarče, hitrost in pospešek tarče. Ključni koraki vključujejo pretvorbo koordinat v fizikalne količine (dolžina, hitrost in pospešek), diskretno aproksimacijo odvodov s končnimi razlikami, ločen izračun za x in y komponento ter kombinacijo komponent v skupno dolžino, hitrost in pospešek. Za natančnejše rezultate bi lahko uporabili interpolacijo za zglajenje poti, statistično obdelavo za zmanjšanje šuma ali Kalmanov filter za boljše sledenje. Hitrost je v tem primeru izražena v pikslih na sekundo (pix/s), za pretvorbo v realne enote pa je potreben kalibracijski faktor (npr. cm/pixel).

Dolžina poti

Dolžino poti tarče izračunamo kot vsoto razdalj med zaporednimi točkami:

$$L = \sum_{i=1}^{n-1} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$$

kjer je:

- L - celotna prehojena pot [pix]
- (x_i, y_i) - koordinate tarče v i -tem okvirju
- n - število meritev

Primer izračuna za prve tri točke iz podanih podatkov:

$$P_1 = (36.56, 370.14)$$

$$P_2 = (44.07, 370.73)$$

$$P_3 = (52.52, 371.23)$$

Izračunamo delno pot:

$$\begin{aligned} L_{1-2} &= \sqrt{(44.07 - 36.56)^2 + (370.73 - 370.14)^2} \\ &= \sqrt{7.51^2 + 0.59^2} = \sqrt{56.58 + 0.35} = 7.55 \text{ pix} \end{aligned}$$

$$\begin{aligned} L_{2-3} &= \sqrt{(52.52 - 44.07)^2 + (371.23 - 370.73)^2} \\ &= \sqrt{8.45^2 + 0.50^2} = \sqrt{71.40 + 0.25} = 8.46 \text{ pix} \end{aligned}$$

$$L_{1-3} = 7.55 + 8.46 = 16.01 \text{ pix}$$

Pretvorbo v realne enote lahko izvedemo, če poznamo kalibracijski faktor (npr. 0.1 cm/pix):

$$L_{\text{real}} = L \times \text{kalibracijski faktor} = 16.01 \times 0.1 = 1.60 \text{ cm}$$

Hitrost in pospešek

Za izračun hitrosti in pospeška tarče iz zaporedja koordinat $\mathbf{r}(t) = (x(t), y(t))$ pri znani frekvenci posnetka ($f_{ps} = 25$), uporabimo naslednje enačbe:

- **Časovni razmik** Čas med posameznimi okvirji:

$$\Delta t = \frac{1}{f_{ps}} = \frac{1}{25} = 0.04 \text{ s}$$

- **Hitrost**

Vektor hitrosti v vsakem časovnem koraku dobimo kot:

$$\mathbf{v}(t) = (v_x(t), v_y(t)) = \left(\frac{x(t + \Delta t) - x(t)}{\Delta t}, \frac{y(t + \Delta t) - y(t)}{\Delta t} \right)$$

Magnitudo hitrosti dobimo kot:

$$v(t) = \sqrt{v_x(t)^2 + v_y(t)^2}$$

- **Pospešek**

Vektor pospeška dobimo kot:

$$\mathbf{a}(t) = (a_x(t), a_y(t)) = \left(\frac{v_x(t + \Delta t) - v_x(t)}{\Delta t}, \frac{v_y(t + \Delta t) - v_y(t)}{\Delta t} \right)$$

Magnitudo pospeška kot:

$$a(t) = \sqrt{a_x(t)^2 + a_y(t)^2}$$

Primer izračuna hitrosti za prva dva okvirja za podane koordinate:

$$\mathbf{r}_1 = (36.56, 370.14)$$

$$\mathbf{r}_2 = (44.07, 370.73)$$

Hitrost:

$$v_x = \frac{44.07 - 36.56}{0.04} = 187.75 \text{ pix/s}$$

$$v_y = \frac{370.73 - 370.14}{0.04} = 14.75 \text{ pix/s}$$

$$v = \sqrt{187.75^2 + 14.75^2} \approx 188.33 \text{ pix/s}$$

Sledenje tarči z Lucas-Kanade postopkom (video1.avi)



Slika 11.4: Sledenje tarče na videu video1.avi, kjer je bila tarča v prvem okvirju *levo* ročno označena z *zelenim* okvirjem, ki pa ga nadalje samodejno premikamo glede na Lukas-Kanade poravnavo med sosednjimi okvirji videa.

11.6 Naloge in vprašanja

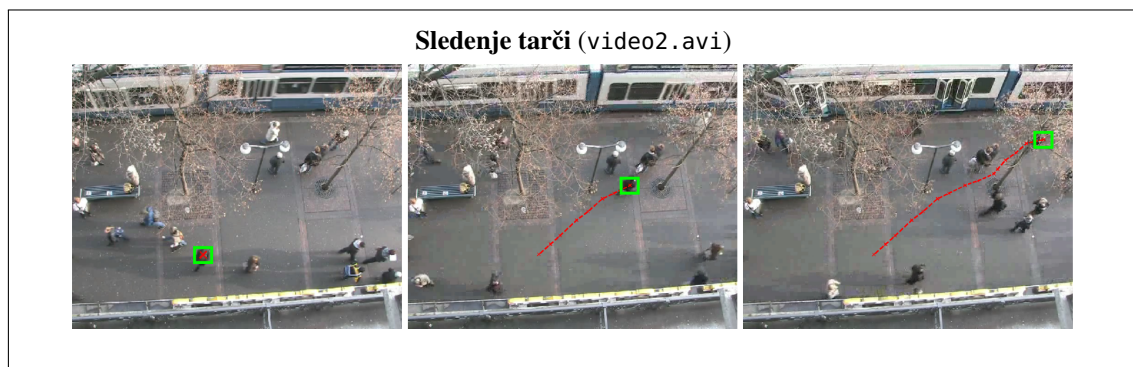
Poročilo naloge pripravite v obliki Python skripte, ki naj izvede zahtevane izračune oz. klice funkcij in izriše pripadajoče slike oz. grafe. Vaša dolžnost je, da v skripti uvozite ustrezne Python knjižnice in dodate morebitne podporne skripte. Skripta se mora izvesti in poustvariti rezultate, kot se zahtevajo v nalogah. Na vprašanja odgovorite v obliki komentarja, naprimer *# Naloga X.1: To je moj odgovor..*

Načrtajte postopek za sledenje tarče na videu video2.avi. V prvem okvirju videa ima tarča središče $(x_c, y_c) = [287, 415]^T$ in se nahaja v pravokotnem oknu $(w, h) = [45, 30]^T$. Za ustrezno sledenje tarči v celotnem videu bo potrebno nadgraditi Lucas-Kanade postopek oz. načrtati nov, bolj robusten postopek sledenja. Pri tem se lahko zgledujete po rešitvah, ki so bile omenjene na predavanjih (namesto kvadratne katera od bolj robustnih različic Lucas-Kanade kriterijske funkcije, odštevanje ozadja, Kalmanov filter, ipd.). Ideje za nadgradnjo postopka lahko naprimer črpate tudi iz preteklih laboratorijskih vaj ali iz poljubnih drugih virov.

Naloga 11.1 Osnovni postopek Lucas-Kanade, ki smo ga uporabili za sledenje tarči v videu video1.avi, ne sledi tarči v celotnem videu video2.avi. Prikažite okvir, kjer postopek odpove in obrazložite zakaj.

Naloga 11.2 Načrtajte izboljššan postopek za sledenje tarče v videu `video2.avi`. Z besedami na kratko opišite vaš izboljšani postopek sledenja tarče. Ocena naloge bo odvisna od inovativnosti vašega postopka sledenja in od tega, kako dolgo in kako natančno vaš postopek sledi tarči.

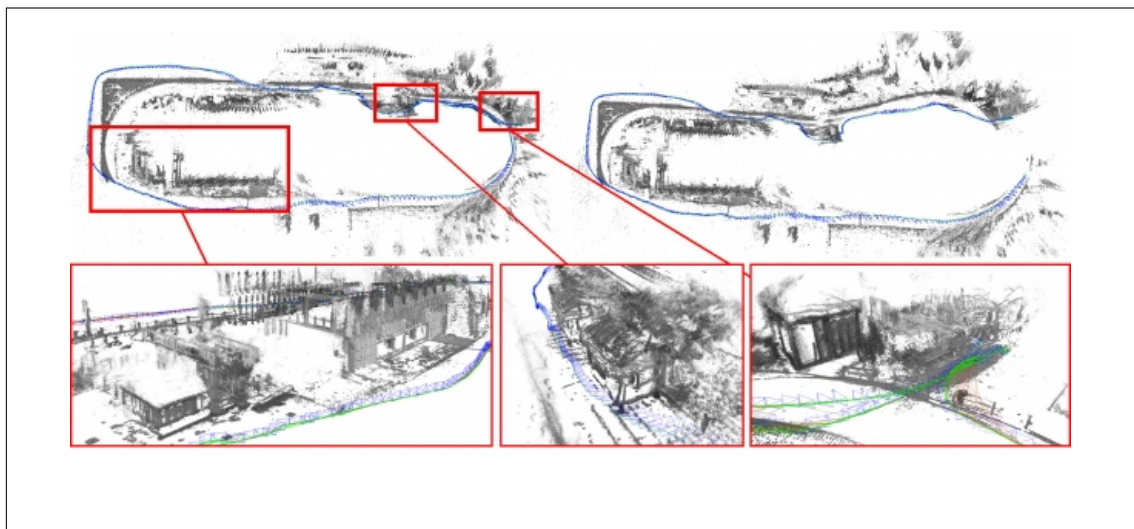
Naloga 11.3 Zaženite vaš postopek sledenja na videu `video2.avi` in določite pot tarče `oPathXY`. Prikažite video in vrišite sled tarče tako, kot je prikazano na sliki 11.5.



Slika 11.5: Zahtevnejši primer sledenje tarči na videu `video2.avi`, kjer je bila tarča v prvem okvirju *levo* ročno označena z *zelenim* okvirjem, ki pa ga nadalje samodejno premikamo glede na *robustno* Lukas-Kanade poravnavo med sosednjimi okvirji videa.

12. Lokalizacija in mapiranje okolja

Če na robota namestimo *tipalo okolice* in ga vozimo po prostoru ter zajemamo meritve *okolice*, lahko na podlagi teh meritev izvedemo hkratno lokalizacijo robota v prostoru in izgradnjo mape tega prostora (slika 12.1). Numerične tehnike, ki to omogočajo, označujemo pod skupnim imenom SLAM¹. Prve tovrstne tehnike so kot meritev okolice uporabljale senzor razdalje (ultrazvočni, laserski (LIDAR), ipd.), v tej vaji pa se bomo posvetili tehnikam, ki temeljijo na vizualni informaciji – sliki zajeti s kamero, ki je na robotu. Te tehnike označujemo tudi kot *Visual SLAM* in zajemajo tudi vizualno odometrijo (VO²) in 3D rekonstrukcijo na podlagi gibanja (SfM³).



Slika 12.1: *Visual SLAM*: primer rekonstrukcije 3D prostora in trajektorije [24].

Pri tej vaji boste na podlagi zaporedja slik zajetih z eno *premično* kamero rekonstruirali 3D

¹SLAM: ang. *Self Localization And Mapping*

²VO: ang. *Visual Odometry*

³SfM: ang. *Structure from Motion*

mapo prostora in določili trajektorijo gibanja kamere v prostoru. S to informacijo lahko kasneje načrtujemo povratno-zančno vodenje robota s kamero tako, da se robot zaveda prostora okoli sebe, npr. aktivno izogiba oviram v prostoru.

12.1 Algoritem stereo vida

Osnovni gradniki postopka *Visual SLAM* so (i) rekonstrukcija 3D scene na podlagi triangulacije, podobno kot pri rekonstrukciji s stereo parom kamer; (ii) ocena esencialne matrike preslikave med dvema slikama; in (iii) analiza esencialne matrike za določanje premika oz. relativne poze kamere (R, T). Če želimo pridobiti metrično rekonstrukcijo 3D prostora je potrebna tudi kalibracija kamere (intrinzični parametri K) in prisotnost kalibracijskih objektov vsaj v enem paru slik, v nasprotnem primeru bo rekonstrukcija 3D prostora skalirana s poljubno skalo (brez kalibracijskega objekta) ali deformirana s poljubno projektivno preslikavo (brez kalibracije kamere).

Triangulacija 3D položaja točke $P^i = [X^i, Y^i, Z^i, 1]^T$ v prostoru je možna na podlagi dveh projekcij te točke $\mathbf{p}_1^i = [u_1^i, v_1^i, 1]^T$ in $\mathbf{p}_2^i = [u_2^i, v_2^i, 1]^T$ na dveh slikah iz dveh pogledov (slika 12.2) s pripadajočimi projekcijskimi matrikami M_1 in M_2 z dimenzijami 3×4 . Za rekonstrukcijo P^i zgradimo linearni sistem enačb:

$$\begin{bmatrix} [\mathbf{p}_1^i]_{\times} M_1 \\ [\mathbf{p}_2^i]_{\times} M_2 \end{bmatrix} \cdot P^i = 0 \rightarrow A \cdot P^i = 0, \quad (12.1)$$

pri čemer ima matrika A dimenzije 6×4 . Notacija $[\mathbf{x}]_{\times}$ predstavlja pretvorbo vektorja $\mathbf{x} = [x_1, x_2, x_3]^T$ v poševno simetrično matriko:

$$[\mathbf{x}]_{\times} = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix}. \quad (12.2)$$

Iščemo netrivialno rešitev sistema (12.1), ki minimizira $\|AP^i\|^2$ ob pogoju $\|P^i\| = 1$. Rešitev dobimo tako, da poiščemo najmanjšo lastno vrednost $A^T A$, pripadajoči lastni vektor pa predstavlja rešitev za P^i . To lahko storimo z uporabo SVD razcepa matrike $A = U\Sigma V^T$, kjer sta matriki U, V unitarni in Σ diagonalna matrika s singularnimi vrednostmi, tako, da poiščemo najmanjšo singularno vrednost in pripadajoči stolpec v matriki V .

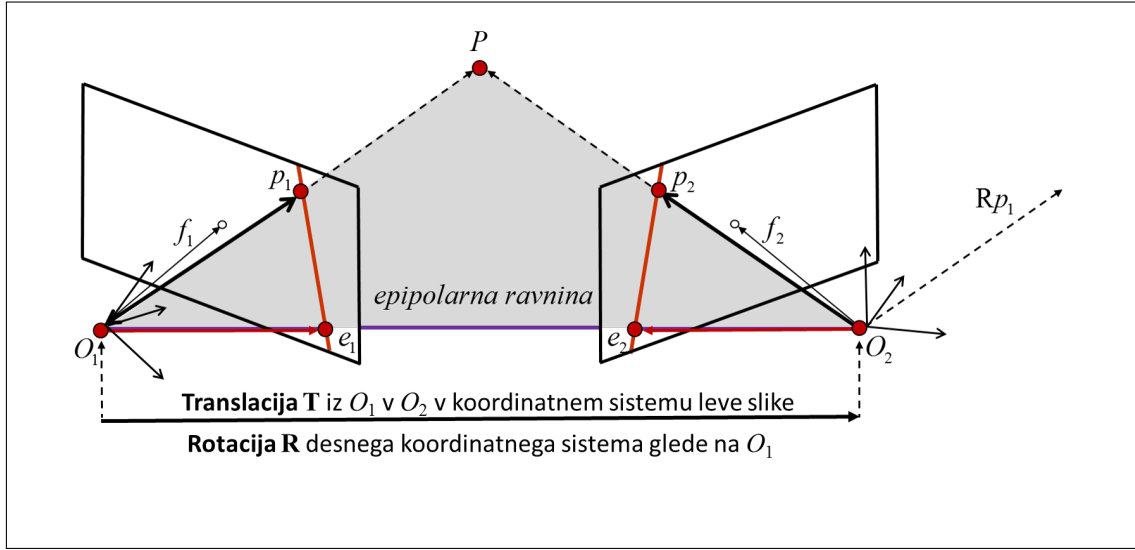
Esencialna matrika E opisuje epipolarno geometrijo kalibriranega stereo sistema kamer tako, da za *normalizirane koordinate* pripadajočih projekcij $\bar{\mathbf{p}}_1 \leftrightarrow \bar{\mathbf{p}}_2$, kjer normaliziramo kot $\bar{\mathbf{p}}_j = K_j^{-1} \mathbf{p}_j$, $j = 1, 2$, velja enačba:

$$\bar{\mathbf{p}}_2^T E \bar{\mathbf{p}}_1 = 0, \quad (12.3)$$

matrika s strukturo $E = [T]_{\times} R$ pa kodira relativen položaj med kamerama. Če normalizirane koordinate nadomestimo z originalnimi dobimo zvezo:

$$(K_2^{-1} \mathbf{p}_2)^T E K_1^{-1} \mathbf{p}_1 = 0 \rightarrow \mathbf{p}_2^T (K_2^{-T} E K_1^{-1}) \mathbf{p}_1 = 0 \rightarrow \mathbf{p}_2^T F \mathbf{p}_1 = 0, \quad (12.4)$$

kjer je F **fundamentalna matrika**. Enačba (12.4) velja tudi v primeru, ko sta kameri nekalibrirani, saj vsebuje nenormalizirane koordinate $\mathbf{p}_1$ in $\mathbf{p}_2$. Enačbi (12.3) in (12.4) predstavljata lastnost epipolarne geometrije, da vektorja $\bar{\mathbf{p}}_1$ in $\bar{\mathbf{p}}_2$ (in tudi $\mathbf{p}_1$ in $\mathbf{p}_2$) ležita v epipolarni ravnini (slika 12.2). Matriki E in F imata dimenzije 3×3 in rang 2, zato bo determinanta $\det E = 0$ in $\det F = 0$.

Slika 12.2: Rekonstrukcija 3D položaja točke P na podlagi dveh projekcij p_1 in p_2 .

12.1.1 Ocenjevanje esencialne matrike E

Esencialno in fundamentalno matriko določimo na enak način z uporabo algoritma **osmih točk**⁴. Za dano množico pripadajočih parov točk $\{\mathbf{p}_1^i \leftrightarrow \mathbf{p}_2^i\}_{i=1,\dots,N}$ lahko enačbo (12.4) zapišemo kot:

$$(\mathbf{p}_1^i \otimes \mathbf{p}_2^i)^T \text{vec}(F) = 0, \text{ kjer je } Q = \begin{bmatrix} (\mathbf{p}_1^1 \otimes \mathbf{p}_2^1)^T \\ \vdots \\ (\mathbf{p}_1^N \otimes \mathbf{p}_2^N)^T \end{bmatrix} \rightarrow Q \cdot \text{vec}(F) = 0. \quad (12.5)$$

Notacija $\text{vec}(F)$ pomeni zapis elementov matrike F v obliki vektorja. Simbol $\otimes$ pa predstavlja Kronekerjev produkt, npr.:

$$\mathbf{p}_1^i \otimes \mathbf{p}_2^i = \begin{bmatrix} u_1^i \cdot p_2^{i^T} & v_1^i \cdot p_2^{i^T} & 1 \cdot p_2^{i^T} \end{bmatrix}^T, \quad (12.6)$$

ki iz dveh trovrstičnih vektorjev (3×1) ustvari devetvrstični vektor (9×1). Za rešitev enačbe (12.5) potrebujemo $N \geq 8$ pripadajočih parov točk. Rešitev dobimo z uporabo SVD razcepa, podobno kot pri triangulaciji, pri čemer pa moramo zagotoviti, da bo $\det F = 0$. To enostavno naredimo tako, da najmanjšo singularno vrednost v $\Sigma = \text{diag}(\{\sigma_j\}_{j=1,2,3})$ postavimo na 0, npr:

$$F = U \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix} V^T \rightarrow F = U \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & 0 \end{bmatrix} V^T. \quad (12.7)$$

Numerično stabilnost zgoraj opisanega algoritma lahko izboljšamo s središčenjem in skaliranjem koordinat $\{\mathbf{p}_1^i\}$ in $\{\mathbf{p}_2^i\}$ tako, da bo njihova srednja vrednost enaka 0, standardna deviacija pa $\sqrt{2}$. Za vsako točko i in množico točk $j = 1, 2$ naredimo preslikavo:

$$\tilde{\mathbf{p}}_j^i = \frac{\sqrt{2}}{\rho_j} (\mathbf{p}_j^i - \mu_j), \text{ oziroma } \tilde{\mathbf{p}}_j^i = \begin{bmatrix} s_j & 0 & -s_j \mu_j^x \\ 0 & s_j & -s_j \mu_j^y \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p}_j^i \rightarrow \tilde{\mathbf{p}}_j^i = T_j \mathbf{p}_j^i. \quad (12.8)$$

kjer sta $\mu_j = \frac{1}{N} \sum_i \mathbf{p}_j^i$ in $\rho_j = \frac{1}{N} \sum_i \|\mathbf{p}_j^i - \mu_j\|^2$. V naslednjem koraku s središčenimi in skaliranimi korespondenčnimi točkami $\{\tilde{\mathbf{p}}_1^i \leftrightarrow \tilde{\mathbf{p}}_2^i\}$ ocenimo fundamentalno matriko $\tilde{F}$ po enačbah (12.5) in

⁴ang. eight-point algorithm

(12.7). Nato kompenziramo središčenje in skaliranje, da dobimo fundamentalno matriko $F = T_2^T \tilde{F} T_1$. Če poznamo tudi intrinzične parametre kamer (K_1 in K_2) lahko glede na (12.4) določimo še esencialno matriko kot $E = K_2^T F K_1$.

12.1.2 Določanje poze (R, T) iz esencialne matrike E

Naj bo svetovni koordinatni sistem v kameri $j = 1$. Potem lahko pripadajočo projekcijsko matriko določimo kot $M_1 = K_1[I|0]$. Relativno pozo (R, T) kamere $j = 2$ bomo določili iz esencialne matrike $E = [T]_{\times} R$, pri čemer obstajajo štiri možne rešitve [4, str. 262]:

$$\begin{aligned} [R_1|T_1] &= [UWV^T|u_3] \\ [R_2|T_2] &= [UW^TV^T|u_3] \\ [R_3|T_3] &= [UWV^T|-u_3] \\ [R_4|T_4] &= [UW^TV^T|-u_3] \end{aligned} \quad (12.9)$$

kjer je:

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (12.10)$$

matrika rotacije za 90° in u_3 zadnji stolpec matrike U v SVD razcepu $E = U\Sigma V^T$. Ker je SVD razcep veljaven tudi če pomnožimo matriki U in V^T z -1, se lahko zgodi, da bo imela katera od rotacijskih matrik $\det R_i = -1$; v tem primeru pomnožimo matriko z -1.

Izmed štirih možnih poz bomo izbrali tisto fizično najbolj smiselno tako, da za vsako od rešitev $[R_k|T_k]$, $k = 1, 2, 3, 4$ s triangulacijo po enačbi (12.1) določimo položaj točk v 3D prostoru $\{P^i\}_k$ in preštejemo vse točke s *pozitivno globino* oz. razdaljo od zaslona. Rešitev k^* z največ takimi točkami je končna rešitev $(R, T) = (R_{k^*}, T_{k^*})$.

12.2 Lokalizacija in mapiranje okolja

Za dan par slik bomo najprej ocenili esencialno matriko E na podlagi izločenih korespondenčnih parov točk $\{\mathbf{p}_1^i \leftrightarrow \mathbf{p}_2^i\}_{i=1,\dots,N}$ in znanih intrinzičnih parametrov kamer K_j , $j = 1, 2$. Nato bomo z analizo E določili relativno pozo (R, T) med kamerama in, glede na kamero $j = 1$, določili projekcijski matriki:

$$M_1 = K_1[I|0] \quad \text{in} \quad M_2 = K_2[R|T]. \quad (12.11)$$

S tema projekcijskima matrikama in triangulacijo točk $\{\mathbf{p}_1^i \leftrightarrow \mathbf{p}_2^i\}_{i=1,\dots,N}$ po enačbi (12.1) bomo zgradili redek oblak 3D točk P^i , ki bodo predstavljale mapo prostora. Z relativno pozo (R, T) kamere $j = 2$ glede na svetovni koordinatni sistem v kameri $j = 1$ je določena tudi lokalizacija kamere v prostoru. Primer lokalizacije in mapiranja okolja na podlagi para slik je prikazan na sliki 12.3.

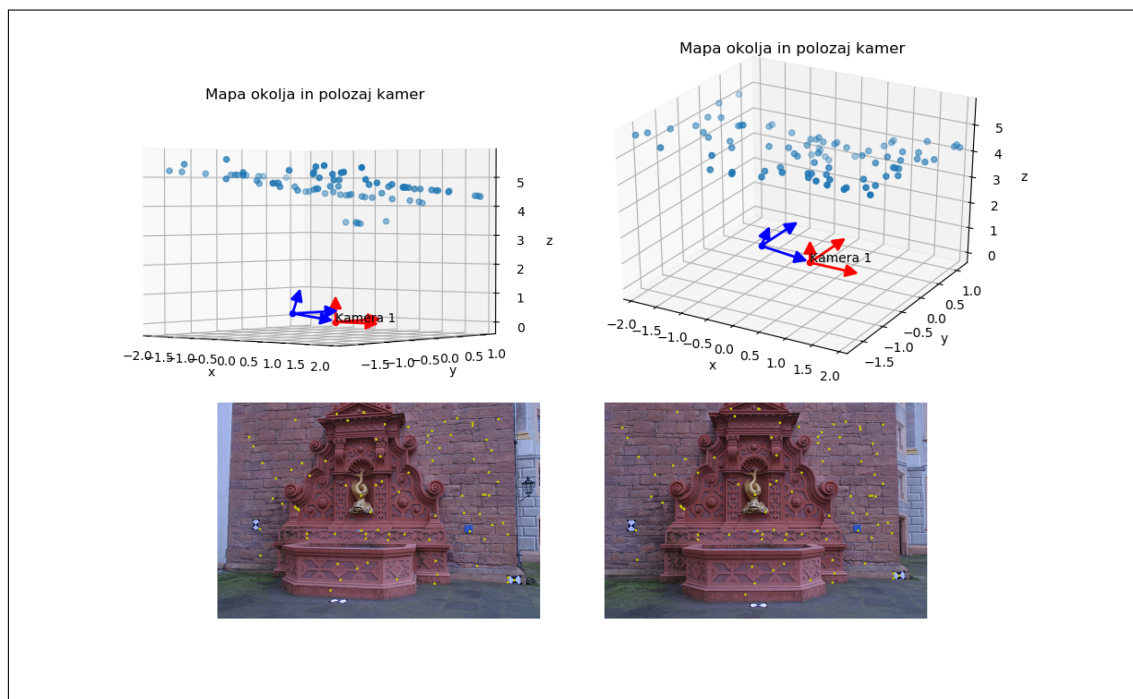
Problem, ki ga pri tej vaji ne obravnavamo je kako poiskati pripadajoče pare točk med dvema slikama, kar potrebujemo za oceno esencialne matrike E . Običajno se poslužujemo detektorjev oslonilnih točk kot so Harrisov detektor oglišč in večnivojskih LoG⁵ in FAST filtrov, okolico oslonilnih točk pa opišemo z deskriptorji SIFT⁶, SURF⁷, ORB⁸, ipd. Pripadajoče pare točk nato določimo s primerjavo deskriptorjev in/ali uporabo postopka RANSAC tako, da večkrat zaporedoma ocenimo E na podlagi naključno izbranih osmih pripadajočih parov točk.

⁵LoG: ang. *Laplacian-of-Gaussian*

⁶SIFT: ang. *Scale Invariant Feature Transform*

⁷SURF: ang. *Speeded-Up Robust Features*

⁸ORB: ang. *Oriented FAST and Rotated BRIEF*



Slika 12.3: Lokacija kamer in 3D mapa prostora kot oblak točk P^i na podlagi pripadajočih točk p_1^i in p_2^i .

12.2.1 Posplošitev na n -pogledov

Kako lokalizirati robota in graditi mapo prostora v primeru več kot dveh pogledov? Opisani postopek lahko zaporedno nadaljujemo in dograjujemo mapo trianguliranih točk ter posodabljammo lokacijo kamere. Ker lahko iste 3D točke vidimo iz več kot dveh pogledov je smiselno narediti skupinski popravek⁹ mape prostora in že določenih poz kamer z nelinearno minimizacijo napake projekcije točk P^i :

$$\arg \min_{\{R_j, T_j\}} \sum_i \sum_j \|M_j(R_j, T_j)P^i - \mathbf{p}_j^i\|^2, \quad (12.12)$$

pri čemer upoštevamo le veljavne projekcije (točka i mora biti vidna na sliki j).

12.3 Vaje z rešitvami

Koda vaje je dostopna na [Git repozitoriju](#). **Gradivo** za vajo vsebuje sliki 0001.jpg in 0002.jpg iste scene, zajete iz dveh pogledov. Dane so tudi koordinate pripadajočih parov oslonilnih točk $\{\mathbf{p}_1^i \leftrightarrow \mathbf{p}_2^i\}_{i=1, \dots, N}$ v dveh datotekah matches0001.txt in matches0002.txt. Obe sliki sta bili zajeti z enako kamero, ki je bila kalibrirana. Intrinzični parametri kamere so:

$$K = \begin{bmatrix} 1379,74 & 0 & 760,35 \\ 0 & 1382,08 & 503,41 \\ 0 & 0 & 1 \end{bmatrix}.$$

Uvozimo Python knjižnice:

```
1 import matplotlib.pyplot as plt
2 from mpl_toolkits.mplot3d import Axes3D
```

⁹ang. *bundle adjustment*

```

3 from matplotlib.patches import FancyArrowPatch
4 from mpl_toolkits.mplot3d import proj3d
5 import numpy as np
6 import PIL.Image as im

```

Vaja 12.1 Napišite funkcijo za triangulacijo pripadajočih parov točk x_1 in x_2 v slikah 1 in 2:

```

1 def linearTriangulation( x1, x2, P1, P2 ):
2     # Tu napišite python kodo funkcije
3     return P

```

kjer so točke x_1 in x_2 podane v homogenih koordinatah v obliki matrik z dimenzijami $3 \times N$, N je število parov točk. Parametra P_1 in P_2 predstavljata 3×4 projekcijski matriki kamere 1 in 2. Funkcija naj vrne matriko P po enačbi (12.1) trianguliranih točk v homogenih koordinatah. Matrika ima dimenzije $4 \times N$.

Preizkusite delovanje triangulacije tako, da ustvarite sintetični projekcijski matriki P_1 in P_2 ter sintetične 3D točke P^* . Nato točke projicirajte v 2D ravnino in z uporabo funkcije `linearTriangulation()` rekonstruirajte točke P . Preverite, da se koordinate sintetičnih in rekonstruiranih točk ujemajo. ■

Najprej pripravimo funkcijo `cross2matrix()`, ki pretvori 3D vektor v antisimetrično (poševno-simetrično) matriko tako, da lahko vektorski produkt izrazimo kot matrično množenje:

```

1 def cross2matrix(x):
2     '''
3     Antisimetricna matrika na podlagi trivrsticnega vektorja
4
5     Vrne antisimetricno matriko M, ki ustreza 3-vektorju x tako, da
6     velja M*y = cross(x,y) za vse 3-vektorje y.
7
8     Parameters
9     -----
10    x : numpy.ndarray
11        Vhodni 3-vektor
12
13    Returns
14    -----
15    oMat : numpy.ndarray
16        3x3 matrika
17    '''
18    return np.array(
19        (
20            (0, -x[2], x[1]),
21            (x[2], 0, -x[0]),
22            (-x[1], x[0], 0)
23        )
24    )

```

Funkcija `linearTriangulation` izvede linearno triangulacijo za rekonstrukcijo 3D točk iz njihovih projekcij v dveh slikah z uporabo projekcijskih matrik. Za vsako točko sestavi linearni sistem enačb $AX = 0$, kjer A vsebuje antisimetrične matrike (izračunane s `cross2matrix`) za

projekcijske enačbe $\mathbf{x}_1 \times (P_1 \mathbf{X}) = \mathbf{0}$ in $\mathbf{x}_2 \times (P_2 \mathbf{X}) = \mathbf{0}$. Rešitev sistema se določi z uporabo SVD razcepa, kjer je 3D točka $\mathbf{X}$ (v homogenih koordinatah) lastni vektor pri najmanjši singularni vrednosti. Rezultat se dehomogenizira z deljenjem z zadnjo koordinato, kar vrne normalizirane 3D koordinate. Funkcija zahteva, da so vhodne točke $\mathbf{x}_1$ in $\mathbf{x}_2$ v homogenih koordinatah (oblika $3 \times N$), projekcijski matriki P_1 in P_2 pa morata biti velikosti 3×4 . Rešitev zapišemo:

```

1 def linearTriangulation(x1, x2, P1, P2):
2     '''
3     Linearna triangulacija
4
5     Parameters:
6     -----
7     x1 : numpy.ndarray
8         Homogene koordinate točk v sliki 1 (3xN)
9     x2 : numpy.ndarray
10        Homogene koordinate točk v sliki 2 (3xN)
11     P1 : numpy.ndarray
12        Projekcijska matrika kamere 1 (3x4)
13     P2 : numpy.ndarray
14        Projekcijska matrika kamere 2 (3x4)
15
16     Returns
17     -----
18     X : numpy.ndarray
19        Homogene koordinate 3D točk (4xN)
20     '''
21     # preveri dimenzije polj
22     assert x1.shape[0] == x2.shape[0], 'Size mismatch of input points'
23     assert x1.shape[1] == x2.shape[1], 'Size mismatch of input points'
24     assert x1.shape[0] == 3, 'Arguments x1, x2 should be 3xN matrices
25         (homogeneous coords)'
26     assert P1.shape == (3, 4), 'Projection matrices should be of size 3x4'
27     assert P2.shape == (3, 4), 'Projection matrices should be of size 3x4'
28
29     num_points = x1.shape[1]
30
31     P = np.zeros((4, num_points))
32
33     # linearni algoritem
34     for j in range(num_points):
35         # sestavi matriko linearnega homogenega sistema enacb
36         A1 = np.dot(cross2matrix(x1[:, j]), P1)
37         A2 = np.dot(cross2matrix(x2[:, j]), P2)
38         A = np.vstack((A1, A2))
39
40         # resi linearni homogeni sistem enacb
41         _, _, vt = np.linalg.svd(A, full_matrices=False)
42         P[:, j] = vt[-1, :]
43
44     # dehomogeniziraj (P je izrazen v homogenih koordinatah)
45     return P / P[-1, :]

```

Preizkus funkcije `linearTriangulation` je zasnovan tako, da najprej generira $N = 10$ naključnih 3D točk $\mathbf{X}$ v homogenih koordinatah (oblika $4 \times N$), kjer so z -koordinate premaknjene v ob-

močje [10, 15] za bolj realistično sceno. Nato se definirajo dve projekcijski matriki P_1 in P_2 kamere, pri čemer ima P_2 dodatni premik v x -smeri (-100), kar simulira stereo konfiguracijo. Projekciji točk $\mathbf{x}_1$ in $\mathbf{x}_2$ v slikah se izračunata kot $\mathbf{x}_1 = P_1 \mathbf{X}$ in $\mathbf{x}_2 = P_2 \mathbf{X}$. Funkcija `linearTriangulation` nato rekonstruira 3D točke $\mathbf{X}_{\text{est}}$ iz projekcij:

```

1 # test linearne triangulacije
2 N = 10 # stevilo 3D točk
3 X = np.random.randn(4, N) # homogene koordinate 3D točk
4 X[2,:] = X[2,:] * 5 + 10
5 X[3,:] = 1
6
7 # projekcijske matrike kamere 1 in 2
8 P1 = np.array((
9     (500, 0, 320, 0),
10    (0, 500, 240, 0),
11    (0, 0, 1, 0)
12 ))
13
14
15 P2 = np.array((
16    (500, 0, 320, -100),
17    (0, 500, 240, 0),
18    (0, 0, 1, 0)
19 ))
20
21
22 # lokacije projiciranih točk v slikah 1 in 2
23 x1 = np.dot(P1, X)
24 x2 = np.dot(P2, X)
25
26 X_est = linearTriangulation(x1, x2, P1, P2)
27
28 print('X_est-X=')
29 print((X_est - X))

```

Končno se izpiše razlika med originalnimi in rekonstruiranimi točkami $\mathbf{X}_{\text{est}} - \mathbf{X}$, kar omogoča preverjanje natančnosti algoritma:

```

X_est-X=
[[ 2.17603713e-14 -4.32986980e-14 -3.19744231e-14  1.06581410e-14
  2.60902411e-14 -5.99520433e-15 -1.47659662e-14 -1.03250741e-14
 -1.44328993e-14  1.82076576e-14]
 [ 1.22679644e-14 -3.44169138e-14 -2.30926389e-14  8.88178420e-15
  1.93040028e-14 -4.55191440e-15 -8.65973959e-15 -4.44089210e-15
 -9.99200722e-15  6.43929354e-15]
 [-6.03961325e-14  7.63833441e-14  7.63833441e-14 -5.32907052e-15
 -3.28626015e-14  9.76996262e-15  4.26325641e-14  1.77635684e-15
  3.19744231e-14  7.10542736e-15]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00]]

```

Razlike v rekonstruiranih koordinatah $\mathbf{X}_{\text{est}}$, na podlagi dveh 2D projekcij, in dejanskimi koordinatami $\mathbf{X}$ so reda 10^{-16} , kar je na nivoju numerične napake za zapis z decimalno vejico tipa `float32`.

Vaja 12.2 Napišite funkcijo za oceno esencialne matrike E z algoritmom osmih točk (enačbe (12.5, 12.7, 12.8)):

```
1 def estimateEssentialMatrix( x1, x2, K1, K2 ):
2     # Tu napišite python kodo funkcije
3     return E
```

kjer so pripadajoči pari točk x_1 in x_2 v slikah 1 in 2 podani v homogenih koordinatah v obliki matrik z dimenzijami $3 \times N$, N je število parov točk. Parametra K_1 in K_2 predstavljata 3×4 matriki intrinzičnih parametrov kamer 1 in 2.

Preizkusite delovanje z uporabo sintetičnih podatkov, ki ste jih ustvarili pri prvi nalogi. Preverite, kakšen vpliv ima na oceno dodajanje Gaussovega šuma projiciranim točkam (s tem simulirate napako detektorjev oslonilnih točk). Ocenite algebrsko napako z izračunom epipolarne omejitve v enačbi (12.3) in geometrijske napake kot srednjo kvadratno razdaljo točk do epipolarne črte:

$$\left[\frac{1}{N} \sum_{i=1}^N (d_{\perp}^2(\mathbf{p}_1^i, \ell_1^i) + d_{\perp}^2(\mathbf{p}_2^i, \ell_2^i)) \right]^{\frac{1}{2}},$$

kjer sta $\ell_2^i = F^T \mathbf{p}_2^i$ in $\ell_1^i = F \mathbf{p}_1^i$ epipolarni črti v slikah 1 in 2, $d_{\perp}(\mathbf{p}, \ell)$ pa meri razdaljo med točko in premico v slikovni ravnini.

Preverite vpliv Gaussovega šuma na algoritem osmih točk z in brez uporabe središčenja in skaliranja koordinat po enačbi (12.8). ■

Najprej pripravimo funkcijo `normalise2dpts()`, ki izvede normalizacijo 2D homogenih točk tako, da prestavi njihov centroid v izhodišče koordinatnega sistema in skalira točke tako, da je njihova povprečna razdalja od izhodišča enaka $\sqrt{2}$. Najprej izračuna centroid μ točk in povprečno standardno deviacijo σ . Nato konstruira transformacijsko matriko T velikosti 3×3 , ki vsebuje skalirni faktor $s = \sqrt{2}/\sigma$ in translacijske komponente za premik centrioda v izhodišče. Kot izhod vrne transformirane točke $T \cdot pts$ in uporabljeno transformacijsko matriko T . Rešitev zapišemo:

```
1 def normalise2dpts(pts):
2     '''
3     Normalizacija 2D homogenih točk
4
5     Funkcija normalizira vsak set točk tako, da bo izhodišče v centroidu in
6     povprečna razdalja do točk od izhodišča sqrt(2)
7
8     Parameters:
9     -----
10    pts : numpy.ndarray
11          Polje homogenih koordinat 2D točk v sliki 1 (3xN)
12
13    Returns
14    -----
15    new_pts : numpy.ndarray
16             Polje preslikanih homogenih koordinat 2D točk v sliki 1 (3xN)
17    T : numpy.ndarray
18        Transformacijska matrika 3x3: newpts = T * pts
19    '''
20    num_points = pts.shape[-1]
21
```

```

22     # centroid
23     mu = np.mean(pts[:2,:], axis=-1).reshape((2,1))
24     # povprečna standardna deviacija
25     sigma = np.mean(np.sqrt(np.sum((pts[:2,:] - mu)**2.0, axis=0)))
26     s = np.sqrt(2) / sigma
27
28     T = np.array(
29         (
30             (s, 0, -s * mu[0][0]),
31             (0, s, -s * mu[1][0]),
32             (0, 0, 1)
33         )
34     )
35
36     return np.dot(T, pts), T

```

Ta normalizacija izboljša numerično stabilnost pri kasnejših izračunih. Algoritem 8-točk za oceno fundamentalne matrike F rešuje linearni sistem enačb $A\mathbf{f} = 0$, kjer $\mathbf{f}$ vsebuje elemente fundamentalne matrike. Za vsak par točk $(x1_i, x2_i)$ zgradi vrstico matrike A z uporabo Kroneckerjevega produkta $\text{kron}(x1_i, x2_i)^T$. Rešitev sistema se najde s SVD razcepom (dodatek A), kjer je F oblikovana iz zadnjega stolpca matrike V^T . Da zagotovi rang 2, funkcija izvede dodatni SVD razcep in nastavi najmanjšo singularno vrednost na 0. Ta postopek zagotovi, da fundamentalna matrika ustreza epipolarnim omejitvam $x2^T F x1 = 0$. Rešitev zapišemo:

```

1  def fundamentalEightPoint(x1, x2):
2      '''
3      Algoritem 8-točk za oceno fundamentalne matrike F
4
5      Rang 3x3 matrike F je v splošnem enak 2, kar ta funkcija zagotavlja preko
6      nastavljanja singularnih vrednosti SVD razcepa. Funkcija ne vključuje
7      normalizacije.
8
9      Referenca: "Multiple View Geometry" (Hartley & Zisserman 2000), Sect. 10.1
10     page 262.
11
12     Parameters:
13     -----
14     x1 : numpy.ndarray
15         Homogene 2D koordinate točk v sliki 1 (3xN)
16     x2 : numpy.ndarray
17         Homogene 2D koordinate točk v sliki 2 (3xN)
18
19     Returns
20     -----
21     F : numpy.ndarray
22         Fundamentalna matrika (3x3)
23     '''
24     # preveri dimenzije vhodnih spremenljivk
25     assert x1.shape == x2.shape, 'Vhodni polji koordinat nista enake velikosti'
26     assert x1.shape[0], 'Vhodne spremenljivke niso 2D točke'
27     assert x1.shape[-1] >= 8, 'Nezadostno število točk za izračun fundamentalne'
28         matrike (potrebujemo >=8)'
29
30     num_points = x1.shape[-1]

```

```

29     # izračunaj matriko A linearnega homogenega sistema, rešitev katere je
      vektor
30     # z elementi fundamentalne matrike
31     A = np.zeros((num_points, 9))
32     for i in range(num_points):
33         A[i, :] = np.kron(x1[:, i], x2[:, i]).T
34
35     # reši linearni homogeni sistem enačb kot  $A \cdot f = 0$ 
36     # korespondence  $x1 \leftrightarrow x2$  so eksaktne ==  $\text{rank}(A)=8$ , torej rešitev obstaja
37     # če so meritve koordinat  $x1, x2$  obremenjene s šumom, potem je  $\text{rank}(A)=9$ ,
      torej ne obstaja
38     # eksaktna rešitev in moramo poiskati rešitev z minimizacijo srednje
      kvadratne napake
39     _, _, Vt = np.linalg.svd(A, full_matrices=False)
40     F = np.reshape(Vt[-1, :], (3, 3)).T
41
42     # vsili  $\det(F)=0$  s projekcijo ocenjene matrike F na množico 3x3 singularnih
      matrik
43     u, s, vt = np.linalg.svd(F)
44     s[2] = 0
45     return np.dot(u, np.dot(np.diag(s), vt))

```

Izboljšana različica algoritma 8-točk vključuje normalizacijo točk pred izračunom fundamentalne matrike. Najprej uporabi funkcijo `normalise2dpts` za normalizacijo točk v obeh slikah, nato izračuna fundamentalno matriko F na normaliziranih točkah z osnovnim algoritmom 8-točk. Na koncu kompenzira normalizacijo z uporabo transformacijskih matrik $T1$ in $T2$ kot $T2^T F T1$. Rešitev zapišemo:

```

1 def fundamentalEightPoint_normalized(x1, x2):
2     '''
3     Oцени fundamentalno matriko glede na dane pripadajoče pare točk in
4     projekcijsko matriko kamere K (intrinzični parametri)
5
6     Parameters:
7     -----
8     x1 : numpy.ndarray
9         Homogene 2D koordinate točk v sliki 1 (3xN)
10    x2 : numpy.ndarray
11        Homogene 2D koordinate točk v sliki 2 (3xN)
12
13    Returns
14    -----
15    F : numpy.ndarray
16        Fundamentalna matrika (3x3)
17    '''
18    # normaliziraj vsak set točk tako, da bo izhodišče v centroidu in
19    # povprečna razdalja do točk od izhodišča  $\sqrt{2}$ 
20    x1_nh, T1 = normalise2dpts(x1)
21    x2_nh, T2 = normalise2dpts(x2)
22
23    # linearna rešitev
24    F = fundamentalEightPoint(x1_nh, x2_nh)
25
26    # kompenzacija normalizacije

```

```
27 | return np.dot(T2.T, np.dot(F, T1))
```

V primeru, da poznamo intrinzične parametre kamer $K1$ in $K2$, lahko ocenimo esencialno matriko E iz parov točk. Postopamo tako, da najprej ocenimo fundamentalno matriko F z normaliziranim algoritmom 8-točk. Nato esencialno matriko izračunamo kot $E = K2^T F K1$, kar predstavlja transformacijo iz prostora normaliziranih slikovnih koordinat v prostor esencialne geometrije. Esencialna matrika E kodira relativno rotacijo in translacijo med kamerama in se uporablja za rekonstrukcijo 3D scene iz več pogledov. Rešitev zapišemo:

```
1 | def estimateEssentialMatrix(x1, x2, K1, K2):
2 |     '''
3 |     Oцени esencialno matriko glede na dane pripadajoče pare točk in
4 |     projekcijsko matriko kamere K (intrinzični parametri)
5 |
6 |     Parameters:
7 |     -----
8 |     x1 : numpy.ndarray
9 |         Homogene 2D koordinate točk v sliki 1 (3xN)
10 |    x2 : numpy.ndarray
11 |        Homogene 2D koordinate točk v sliki 2 (3xN)
12 |    K1 : numpy.ndarray
13 |        Projekcijska matrika kamere 1 (3x4)
14 |    K2 : numpy.ndarray
15 |        Projekcijska matrika kamere 2 (3x4)
16 |
17 |    Returns
18 |    -----
19 |    E : numpy.ndarray
20 |        Esencialna matrika (3x3)
21 |    '''
22 |    # oceni fundamentalno matriko  $K2^{-T} * F * K1^{-1}$ 
23 |    F = fundamentalEightPoint_normalized(x1, x2)
24 |
25 |    # izračunaj esencialno matriko iz ocenjene fundamentalne matrike
26 |    return np.dot(K2.T, np.dot(F, K1))
```

Funkcija `distPoint2EpipolarLine()` izračuna povprečno razdaljo med točkami in njihovimi epipolarnimi črtami, kar služi kot mera napake pri oceni fundamentalne matrike F . Najprej združi vse točke iz obeh slik ($x1$ in $x2$) v homogenem formatu, nato pa izračuna epipolarne črte za vsako točko: za točke v prvi sliki uporabi $F^T x2$, za točke v drugi sliki pa $F x1$. Razdaljo med točko in epipolarno črto izračuna po formuli za razdaljo točke do premice v homogenih koordinatah $\frac{|ax+by+c|}{\sqrt{a^2+b^2}}$, kjer so a , b in c koeficienti epipolarne črte. Končni rezultat je povprečna kvadratna razdalja, normalizirana s številom točk, kar omogoča kvantificiranje napake pri ujemanju epipolarnih omejitev $x2^T F x1 = 0$. Rešitev zapišemo:

```
1 | def distPoint2EpipolarLine(F, x1, x2):
2 |     '''
3 |     Izračunaj razdaljo med točko in epipolarno crto
4 |
5 |     Parameters:
6 |     -----
7 |     F : numpy.ndarray
8 |         Fundamentalna matrika
```



```

9     x1 : numpy.ndarray
10         Homogene 2D koordinate točk v sliki 1 (3xN)
11     x2 : numpy.ndarray
12         Homogene 2D koordinate točk v sliki 2 (3xN)
13
14     Returns
15     -----
16     dist2 : float
17         vsota kvadriranih razdalj od točk do epipolarne crte, normalizirana s
18         številom koordinat točk
19
20     num_points = x1.shape[-1]
21
22     homog_points = np.hstack((x1, x2))
23     epi_lines = np.hstack((np.dot(F.T, x2), np.dot(F, x1)))
24
25     denom = epi_lines[0,:]**2.0 + epi_lines[1,:]**2.0
26     return np.sqrt(
27         np.sum(
28             (np.sum(epi_lines*homog_points, axis=0)**2.0) / denom
29         )

```

Testna koda ocenjuje delovanje funkcij `fundamentalEightPoint()` in `fundamentalEightPoint_normalized()` na simuliranih podatkih. Najprej generira $N = 40$ naključnih 3D točk $\mathbf{X}$ v homogenih koordinatah, ki jih postavi v realističen območje (z -koordinate med 5 in 15). Dve kameri s projekcijskima matrikama $\mathbf{P}_1$ in $\mathbf{P}_2$ (druga kamera premaknjena za 100 enot v x -smeri) projicirata 3D točke v dve sliki, kar da točke $\mathbf{x}_1$ in $\mathbf{x}_2$. Test primerja tri scenarije: (1) idealne korespondence brez šuma, (2) koordinate z dodanim Gaussovim šumom ($\sigma = 0.1$), in (3) normalizirane koordinate s šumom. Za vsak scenarij izračuna fundamentalno matriko $\mathbf{F}$ in ovrednoti njeno natančnost z dvema metrikama: algebrasko napako (norma epipolarne omejitve $\mathbf{x}_2^T \mathbf{F} \mathbf{x}_1$) in geometrično napako (povprečna razdalja točk do epipolarnih črt). Testno kodo zapišemo:

```

1  # test algoritma 8-točk
2  N = 40 # stevilo 3D točk
3  X = np.random.randn(4,N) # homogene koordinate 3D točk
4
5  # postavitev simulirane scene z eksaktnimi korespondencami
6  X[2,:] = X[2,:] * 5 + 10
7  X[3,:] = 1
8
9  # projekcijske matrike kamere 1 in 2
10 P1 = np.array((
11     (500, 0, 320, 0),
12     (0, 500, 240, 0),
13     (0, 0, 1, 0)
14 ))
15
16 P2 = np.array((
17     (500, 0, 320, -100),
18     (0, 500, 240, 0),
19     (0, 0, 1, 0)
20 ))
21 )

```

```

22
23 # lokacije projiciranih točk v slikah 1 in 2
24 x1 = np.dot(P1, X)
25 x2 = np.dot(P2, X)
26
27 sigma = 1e-1
28 noisy_x1 = x1 + sigma * np.random.randn(x1.shape[0], x1.shape[1], x1.shape[2])
29 noisy_x2 = x2 + sigma * np.random.randn(x1.shape[0], x1.shape[1], x1.shape[2])
30
31 # ocena fundamentalne matrik z algoritmom 8-točk
32 F = fundamentalEightPoint(x1, x2)
33
34 # preveri epipolarno omejitev  $x2(i)' * F * x1(i) = 0$  za vse točke  $i$ .
35 err_algebraic = np.linalg.norm(np.sum(x2*np.dot(F, x1), axis=0)) / np.sqrt(N)
36 err_dist_epi_line = distPoint2EpipolarLine(F, x1, x2)
37
38 print('Eksaktne korespondence')
39 print('Algebrajska napaka: ', err_algebraic)
40 print('Geometrična napaka: ', err_dist_epi_line, ' px')
41
42 % # test algoritma 8-točk s pošumljenimi koordinatami točk
43 F = fundamentalEightPoint(noisy_x1, noisy_x2)
44
45 # preveri epipolarno omejitev  $x2(i)' * F * x1(i) = 0$  za vse točke  $i$ .
46 err_algebraic = np.linalg.norm(np.sum(noisy_x2*np.dot(F, noisy_x1), axis=0)) /
    np.sqrt(N)
47 err_dist_epi_line = distPoint2EpipolarLine(F, noisy_x1, noisy_x2)
48
49 print(f'\nKorespondence z Gaussovim šumom (sigma={sigma}) na koordinatah,
    algoritem 8-točk')
50 print('Algebrajska napaka: ', err_algebraic)
51 print('Geometrična napaka: ', err_dist_epi_line, ' px')
52
53 # test normaliziranega algoritma 8-točk s pošumljenimi koordinatami točk
54 Fn = fundamentalEightPoint_normalized(noisy_x1, noisy_x2)
55
56 err_algebraic = np.linalg.norm(np.sum(noisy_x2*np.dot(Fn, noisy_x1), axis=0)) /
    np.sqrt(N)
57 err_dist_epi_line = distPoint2EpipolarLine(Fn, noisy_x1, noisy_x2)
58
59 print(f'\nKorespondence z Gaussovim šumom (sigma={sigma}) na koordinatah,
    normaliziran algoritem 8-točk')
60 print('Algebrajska napaka: ', err_algebraic)
61 print('Geometrična napaka: ', err_dist_epi_line, ' px')

```

Testna koda vrne naslednji rezultat:

Eksaktne korespondence
 Algebrajska napaka: 2.5228702197057964e-09
 Geometrična napaka: 4.069852295117214e-10 px

Korespondence z Gaussovim šumom (sigma=0.1) na koordinatah, algoritem 8-točk
 Algebrajska napaka: 31.75717454596762
 Geometrična napaka: 622.2465781038167 px

Korespondence z Gaussovim šumom ($\sigma=0.1$) na koordinatah, normaliziran
 $\rightarrow$ algoritem 8-točk
 Algebrajska napaka: 0.0046674954809248805
 Geometrična napaka: 38.303436435518336 px

Rezultati prikazujejo, kako bistveno normalizacija pred izvedbo algoritma 8-točk izboljša robustnost algoritma ob prisotnosti šuma, kar je ključno za praktične aplikacije računalniškega vida. Opazimo tudi, da je geometrična napaka v primerjavi z algebrajsko bistveno bolj občutljiva na napake prileganja z algoritmom 8-točk. Algebrajsko napako namreč direktno minimizira algoritem 8-točk, medtem ko je geometrična napaka nepristranska ocena.

Vaja 12.3 Napišite funkcijo za oceno poze na podlagi esencialne matrike E :

```
1 def decomposeEssentialMatrix( E ):
2     # Tu napišite python kodo funkcije
3     return R, u3
```

kjer je E vhodna 3×3 esencialna matrika. Po enačbi (12.9) naj funkcija ustvari in vrne štiri možne rešitve poze (R_k, T_k) , $k = 1, 2, 3, 4$, pri čemer vrne dve možni rotacijski matriki UWV^T in UW^TV^T kot matriko $3 \times 3 \times 2$ v spremenljivki R in translacijo kot 3×1 vektor v spremenljivki $u3$. Zagotovite, da imajo rotacijske matrike determinanto enako 1 in vektor $u3$ normo enako 1.

Funkcija `decomposeEssentialMatrix()` razcepi esencialno matriko E na možne rotacije in translacije med kamerama. Najprej izvede singularni razcep (SVD) matrike $E = U\mathbf{\Sigma}V^T$, kjer je translacijski vektor $\mathbf{u}_3$ zadnji stolpec matrike U . Rotacijske matrike se izračunajo kot $R_1 = UWV^T$ in $R_2 = UW^TV^T$, kjer je W preddefinirana matrika oblike:

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Funkcija zagotovi, da imajo rotacijske matrike determinanto $+1$ (pravilna orientacija), in normalizira translacijski vektor na enotsko dolžino. Rezultat sta dve možni rotaciji ($3 \times 3 \times 2$) in en translacijski vektor $\mathbf{u}_3$. Rešitev zapišemo:

```
1 def decomposeEssentialMatrix(E):
2     '''
3     Za dano esencialno matriko izračunaj gibanje kamere, tj. R in T tako, da bo
4     E ~ T_x R
5     function [R,u3] = decomposeEssentialMatrix(E)
6
7     Parameters:
8     -----
9     E : numpy.ndarray
10         Esencialna matrika (3x3)
11
12     Returns
13     -----
14     R : numpy.array
15         Polje (3x3x2) dveh možnih rotacij
16     u3 : numpy.array
17         Vektor s translacijo
18     '''
19     U, _, Vt = np.linalg.svd(E)
```

```

19
20     # translacija
21     u3 = U[:,2]
22
23     # rotacija
24     R = np.zeros((3,3,2))
25     W = np.array(((0, -1, 0), (1, 0, 0), (0, 0, 1)))
26     R[:, :, 0] = np.dot(U, np.dot(W, Vt))
27     R[:, :, 1] = np.dot(U, np.dot(W.T, Vt))
28
29     if np.linalg.det(R[:, :, 0]) < 0:
30         R[:, :, 0] = -R[:, :, 0]
31
32     if np.linalg.det(R[:, :, 1]) < 0:
33         R[:, :, 1] = -R[:, :, 1]
34
35     if np.linalg.norm(u3) != 0:
36         u3 = u3 / np.linalg.norm(u3)
37
38     return R, u3

```

Funkcijo bomo preizkusili pri naslednji vaji 12.4.

Vaja 12.4 Napišite funkcijo za določanje končne poze (R, T):

```

1 def disambiguateRelativePose( R, u3, x1, x2, K1, K2 ):
2     # Tu napišite python kodo funkcije
3     return R, u3

```

kjer so vhodni parametri R možni rotacijski matriki, $u3$ možna translacija (± 1), $x1$ in $x2$ točke podane v homogenih koordinatah v obliki matrik z dimenzijami $3 \times N$ (N je število parov točk), $K1$ in $K2$ pa predstavljata 3×4 matriki intrinzičnih parametrov kamer 1 in 2. S triangulacijo določite položaj točk v prostoru P^i in preštejte koliko točk ima pozitivno razdaljo od zaslona ene in druge kamere – rotacija in translacija pri kateri je število teh točk maksimalno naj bo izhodna poza (R, T).

Preizkusite funkciji z danimi pripadajočimi pari točk na danih slikah in danimi intrinzičnimi parametri kamere K , ki je enaka za obe sliki. S triangulacijo (12.1) rekonstruirajte 3D položaj danih pripadajočih parov točk in jih narišite v 3D sceno. Vrišite tudi 3D položaj prve in druge kamere. Rezultat primerjajte s sliko 12.3. ■

Funkcija `disambiguateRelativePose()` izbere pravilno kombinacijo rotacije in translacije izmed štirih možnosti (dve rotaciji $\times$ dve smeri translacije) tako, da rekonstruirane 3D točke ležijo pred obema kamerama. Za vsako kombinacijo sestavi projekcijsko matriko druge kamere $\mathbf{M}_1 = \mathbf{K}_1[\mathbf{R}|\mathbf{t}]$ in uporabi linearno triangulacijo za rekonstrukcijo 3D točk $\mathbf{P}_{C0}$ v koordinatnem sistemu prve kamere. Nato preveri, koliko točk ima pozitivno globino (z -koordinato) v obeh kamerah ($\mathbf{P}_{C0}$ in $\mathbf{P}_{C1} = \mathbf{R}\mathbf{P}_{C0} + \mathbf{t}$). Kombinacija z največjim številom točk pred kamerami se izbere kot končna rešitev. Ta postopek rešuje dvoumnost v interpretaciji esencialne matrike in zagotavlja fizikalno smiselno rekonstrukcijo. Rešitev zapišemo:

```

1 def disambiguateRelativePose(Rots, u3, points0_h, points1_h, K0, K1):
2     '''
3     Poišči pravilno relativno pozo kamere (med štirimi možnimi) in vrni tisto,
        pri

```

```

4     kateri točke ležijo pred slikovno ravnino (s pozitivno globino)
5
6     Parameters:
7     -----
8     Rots : numpy.array
9           Polje (3x3x2) dveh možnih rotacij
10    u3 : numpy.array
11         Vektor s translacijo
12    x1 : numpy.ndarray
13         Homogene koordinate točk v sliki 1 (3xN)
14    x2 : numpy.ndarray
15         Homogene koordinate točk v sliki 2 (3xN)
16    K1 : numpy.ndarray
17         Kalibracijska matrika kamere 1 (3x3)
18    K2 : numpy.ndarray
19         Kalibracijska matrika kamere 2 (3x3)
20
21     Returns
22     -----
23     R : numpy.array
24         Polje (3x3) rotacije
25     T : numpy.array
26         Vektor s translacijo
27
28         kjer  $[R|t] = T_{C1\_C0} = T_{C1\_W}$  predstavlja transformacijo točk iz
29         svetovnega koordinatnega sistema (enak tistemu od kamere 1) v kamero 2
30     '''
31     # projekcijska matrika kamere 1
32     M0 = np.dot(K0, np.eye(3,4))
33
34     total_points_in_front_best = 0
35     for iRot in (0,1):
36         R_C1_C0_test = Rots[:, :, iRot]
37
38         for iSignT in (1,2):
39             T_C1_C0_test = (u3 * (-1)**iSignT).reshape((3, 1))
40
41             M1 = np.dot(K1, np.hstack((R_C1_C0_test, T_C1_C0_test)))
42
43             P_C0 = linearTriangulation(points0_h, points1_h, M0, M1)
44
45             # projekcija v obe kameri
46             P_C1 = np.dot(np.hstack((R_C1_C0_test, T_C1_C0_test)), P_C0)
47
48             num_points_in_front0 = np.sum(P_C0[2, :] > 0)
49             num_points_in_front1 = np.sum(P_C1[2, :] > 0)
50             total_points_in_front = num_points_in_front0 + num_points_in_front1
51
52             if (total_points_in_front > total_points_in_front_best):
53                 # shrani rotacijo, ki vrne največje število točk pred obema
54                 # kamerama
55                 R = R_C1_C0_test
56                 T = T_C1_C0_test
57                 total_points_in_front_best = total_points_in_front

```

```
58 |         return R, T
```

Testna koda implementira celoten potek strukturne rekonstrukcije scene iz dveh slik. Najprej naloži dve vhodni sliki (0001.jpg in 0002.jpg) ter kalibracijsko matriko $\mathbf{K}$ kamere, ki vsebuje intrinzične parametre (goriščne razdalje in optično središče). Ujemanja točk med slikama se naložijo iz datotek matches0001.txt in matches0002.txt ter pretvorijo v homogene koordinate. Postopek rekonstrukcije vključuje naslednje ključne korake: (1) oceno esencialne matrike $\mathbf{E}$ z normaliziranim algoritmom 8-točk, (2) razcep esencialne matrike na možne rotacije in translacije s funkcijo `decomposeEssentialMatrix()`, (3) razrešitev dvoumnosti relativne poze s funkcijo `disambiguateRelativePose()`, ki izbere tisto kombinacijo rotacije $\mathbf{R}$ in translacije $\mathbf{t}$, kjer večina rekonstruiranih točk leži pred obema kamerama, in (4) linearno triangulacijo 3D točk. Rezultati se vizualizirajo v 3D prostoru, kjer so prikazane rekonstruirane točke scene ter koordinatni sistemi obeh kamer (rdeča za prvo kamero, modra za drugo), skupaj z ustreznimi 2D prikazi slik z označenimi ujemanji.

Vizualizacijski del uporablja razred `Arrow3D` za prikaz 3D koordinatnih sistemov kamer, ki deduje od razreda `FancyArrowPatch` in implementira metode za pravilno projiciranje v 2D prostor. Funkcija `plotCoordinateFrame()` riše koordinatne osi kot 3D puščice. 3D točke scene se prikažejo s funkcijo `scatter()`, medtem ko se 2D prikazi originalnih slik z označenimi ujemanji pripravijo z `imshow()` in `plot()`. Celoten postopek omogoča preverjanje kakovosti rekonstrukcije in pravilnosti delovanja vseh vključenih funkcij, od ocene esencialne matrike do končne triangulacije in vizualizacije.

Testno kodo zapišemo:

```
1 | img_1 = np.array(im.open('0001.jpg'))
2 | img_2 = np.array(im.open('0002.jpg'))
3 |
4 | K = np.array(
5 |     (
6 |         (1379.74, 0, 760.35),
7 |         (0, 1382.08, 503.41),
8 |         (0, 0, 1)
9 |     )
10 | )
11 |
12 | p1 = np.genfromtxt('matches0001.txt')
13 | p2 = np.genfromtxt('matches0002.txt')
14 |
15 | p1 = np.vstack((p1, np.ones((1, p1.shape[-1]))))
16 | p2 = np.vstack((p2, np.ones((1, p2.shape[-1]))))
17 |
18 | # oceni esencialno matriko z algoritmom 8-točk
19 | E = estimateEssentialMatrix(p1, p2, K, K)
20 |
21 | # izloči relativen položaj kamere (R, T) iz esencialne matrike
22 | Rots, u3 = decomposeEssentialMatrix(E)
23 |
24 | # najdi rešitev med 4-imi možnimi
25 | R_C2_W, T_C2_W = disambiguateRelativePose(Rots, u3, p1, p2, K, K)
26 |
27 | # izvedi triangulacijo oblaka točk s končno preslikavo (R, T)
28 | M1 = np.dot(K, np.eye(3, 4))
29 | M2 = np.dot(K, np.hstack((R_C2_W, T_C2_W)))
30 | P = linearTriangulation(p1, p2, M1, M2)
```



```

31
32 # vizualiziraj 3d sceno
33 class Arrow3D(FancyArrowPatch):
34     def __init__(self, xs, ys, zs, *args, **kwargs):
35         super().__init__((0, 0), (0, 0), *args, **kwargs)
36         self._verts3d = xs, ys, zs
37
38     def do_3d_projection(self, renderer=None):
39         xs3d, ys3d, zs3d = self._verts3d
40         xs, ys, zs = proj3d.proj_transform(xs3d, ys3d, zs3d, self.axes.M)
41         self.set_positions((xs[0], ys[0]), (xs[1], ys[1]))
42         return np.min(zs) # Return the zorder value
43
44     def draw(self, renderer):
45         super().draw(renderer)
46
47 def plotCoordinateFrame(ax, rotation, origin, color):
48     for i in range(3):
49         lines = [[float(origin[j]), float(origin[j] + rotation[j,i])] for j in
50                 range(3)]
51         a = Arrow3D(lines[0], lines[1], lines[2], mutation_scale=20,
52                    lw=2, arrowstyle="->", color=color)
53         ax.add_artist(a)
54
55 fig = plt.figure()
56 ax = fig.add_subplot(111, projection='3d')
57
58 # nariši točke okolja
59 ax.scatter(P[0, :], P[1, :], P[2, :])
60
61 # nariši položaj kamer
62 plotCoordinateFrame(ax, np.eye(3), np.zeros((3,1)), 'r')
63 center_cam2_W = -np.dot(R_C2_W.T, T_C2_W)
64 plotCoordinateFrame(ax, R_C2_W.T, center_cam2_W, 'b')
65 ax.scatter(0, 0, 0, c='r')
66 ax.scatter(center_cam2_W[0], center_cam2_W[1], center_cam2_W[2], c='b')
67
68 ax.text(0, 0, 0, 'Kamera 1')
69 # ax.text(center_cam2_W[0], center_cam2_W[1], center_cam2_W[2], 'Kamera 2')
70
71 ax.set_xlabel('x')
72 ax.set_ylabel('y')
73 ax.set_zlabel('z')
74 ax.set_title('Mapa okolja in položaj kamer')
75
76 plt.axis('equal')
77 plt.show()
78
79 plt.figure()
80 plt.subplot(1,2,1)
81 plt.imshow(img_1)
82 plt.axis('off')
83 plt.plot(p1[0, :], p1[1, :], marker='.', lw=0, ms=8, mfc=None, c='y', mew=1,
84         mec='y')
85 plt.subplot(1,2,2)
86 plt.imshow(img_2)

```

```
85 plt.axis('off')
86 plt.plot(p2[0, :], p2[1, :], marker='.', lw=0, ms=8, mfc=None, c='y', mew=1,
           mec='y')
87 plt.plot()
```

Rešitev je prikazana na sliki 12.3.

12.4 Naloge in vprašanja

Poročilo naloge pripravite v obliki Python skripte, ki naj izvede zahtevane izračune oz. klice funkcij in izriše pripadajoče slike oz. grafe. Vaša dolžnost je, da v skripti uvozite ustrezne Python knjižnice in dodate morebitne podporne skripte. Skripta se mora izvesti in poustvariti rezultate, kot se zahtevajo v nalogah. Na vprašanja odgovorite v obliki komentarja, naprimer *# Naloga X.1: To je moj odgovor..*

Naloga 12.1 Preizkusite delovanje algoritmov na slikah, ki jih zajamete z lastnim fotoaparatom. Pred tem kalibrirajte fotoaparat tako, da večkrat slikate šahovnico in nato določite intrinzične parametre kamere s funkcijami v knjižnici OpenCV (dodatne naloge pri vaji 8). Nastavitev fotoaparata nato ne spreminjajte (enak zoom in fokus – izklopite avtofokus). Zajemite več slik prostora in rekonstruirajte 3D okolje kot oblak točk. Ali dobite smiselno rešitev za lokalizacijo (položaj kamere med zajemom) in okolja?

Naloga 12.2 Rešitev, ki ste jo dobili lahko še izboljšate s skupinskim popravkom (ang. *bundle adjustment*), pri čemer se lahko zgledujete po naslednji rešitvi:

http://scipy-cookbook.readthedocs.io/items/bundle_adjustment.html



13. Vizualna kontrola kakovosti

Vizualna kontrola kakovosti je proces razpoznavanja vzorcev na podlagi vizualne oz. slikovne informacije, ki je namenjen podpori pri odločanju o ustreznosti oz. neustreznosti objektov zanimanja. Interpretacija surovih slikovnih podatkov je pogosto zelo zamudna, naporna in včasih tudi zelo zahtevna naloga, zato je smiselno razvijati računalniške postopke za podporo pri odločanju, ki temeljijo na avtomatski analizi, opisovanju in razpoznavanju opazovanih objektov. Splošni sistem za razpoznavanje vzorcev lahko obravnavamo kot zaporedje šestih osnovnih funkcionalnih pod-sistemov: **zajemanje podatkov, obdelava in obnova zajetih podatkov, razgradnja obnovljenih podatkov, izločanje značilnic, razvrščanje ter vrednotenje razvrščanja** (slika 13.1). Pri vaji se bomo spoznali z osnovno razgradnjo slik, izločanjem značilnic in razvrščanjem za problem vizualne kontrole kakovosti farmacevtskih tablet.

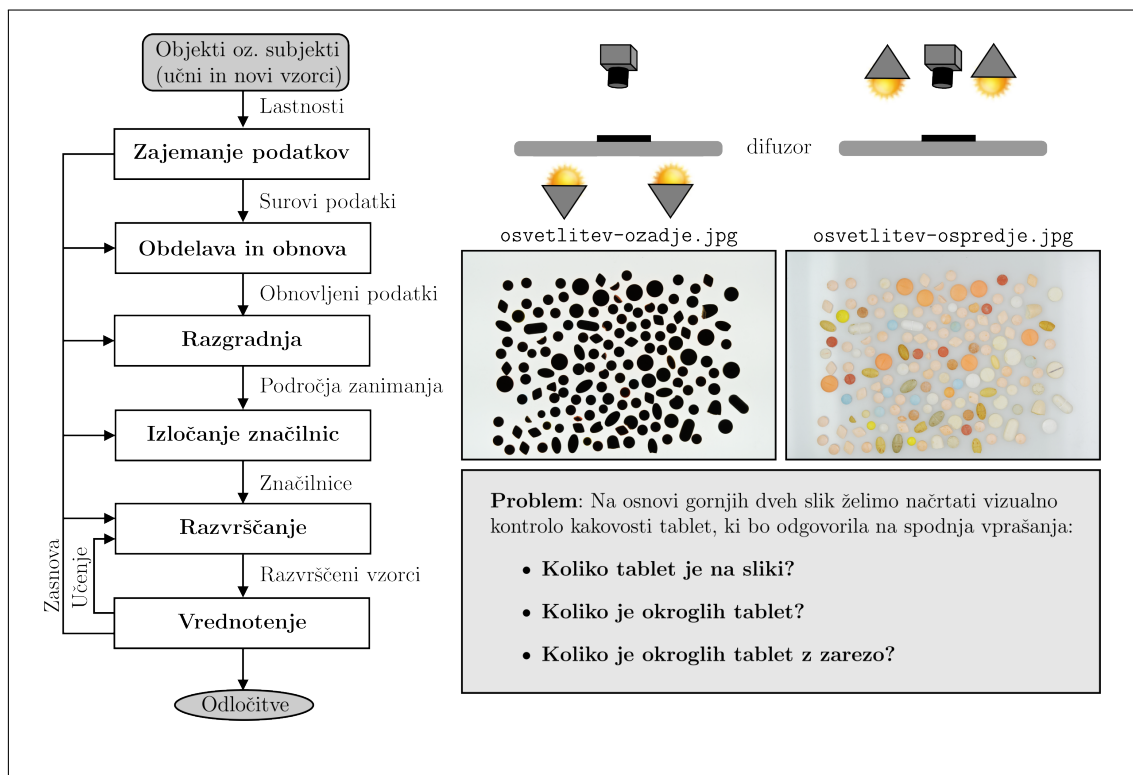
13.1 Sistem za razpoznavanje vzorcev

13.1.1 Razgradnja objektov zanimanja

Razgradnja¹ združuje postopke, s katerimi sliko razdelimo na osnovna področja oziroma objekte. Razgradnjo bomo izvedli z upravljanja slike z osvetlitvijo iz ozadja in z označevanjem objektov. Pri označevanju objektov vsakemu (binarnemu) objektu na sliki priredimo lastno oznako oz. kodo. Primer izvajanja upravljanja:

```
1 import cv2
2 import numpy as np
3
4 # Naložimo sliko v sivinski lestvici
5 image = cv2.imread("tablet.png", cv2.IMREAD_GRAYSCALE)
6
7 # Nastavimo prag
8 _, thresholded = cv2.threshold(image, 128, 255, cv2.THRESH_BINARY)
9
10 # Prikaz rezultatov
```

¹razgradnja: ang. image segmentation



Slika 13.1: Sistem za razpoznavanje vzorcev (levo) in izziv analize slik tablet na vaji (desno).

```

11 cv2.imshow("Thresholded Image", thresholded)
12 cv2.waitKey(0)
13 cv2.destroyAllWindows()

```

13.1.2 Analiza objektov

Analizo objektov v razgrajeni sliki izvedemo z izločanjem značilnic oz. kvantitativnimi opisi lastnosti posameznih področij slike. Posamezno področje slike lahko predstavimo z **zunanjimi lastnostmi** (mejami področja) in/ali **notranjimi lastnostmi** (slikovnimi elementi področja). Zunanje lastnosti omogočajo predstavitev in opisovanje področij z izločanjem značilnic iz mej področij, npr. obseg, usmeritev, največja dolžina, gladkost meje, itd. Notranje lastnosti po drugi strani omogočajo predstavitev in opisovanje pojavnosti področij, kot so npr. homogenost, barva, tekstura, itd.

Zunanje lastnosti področja

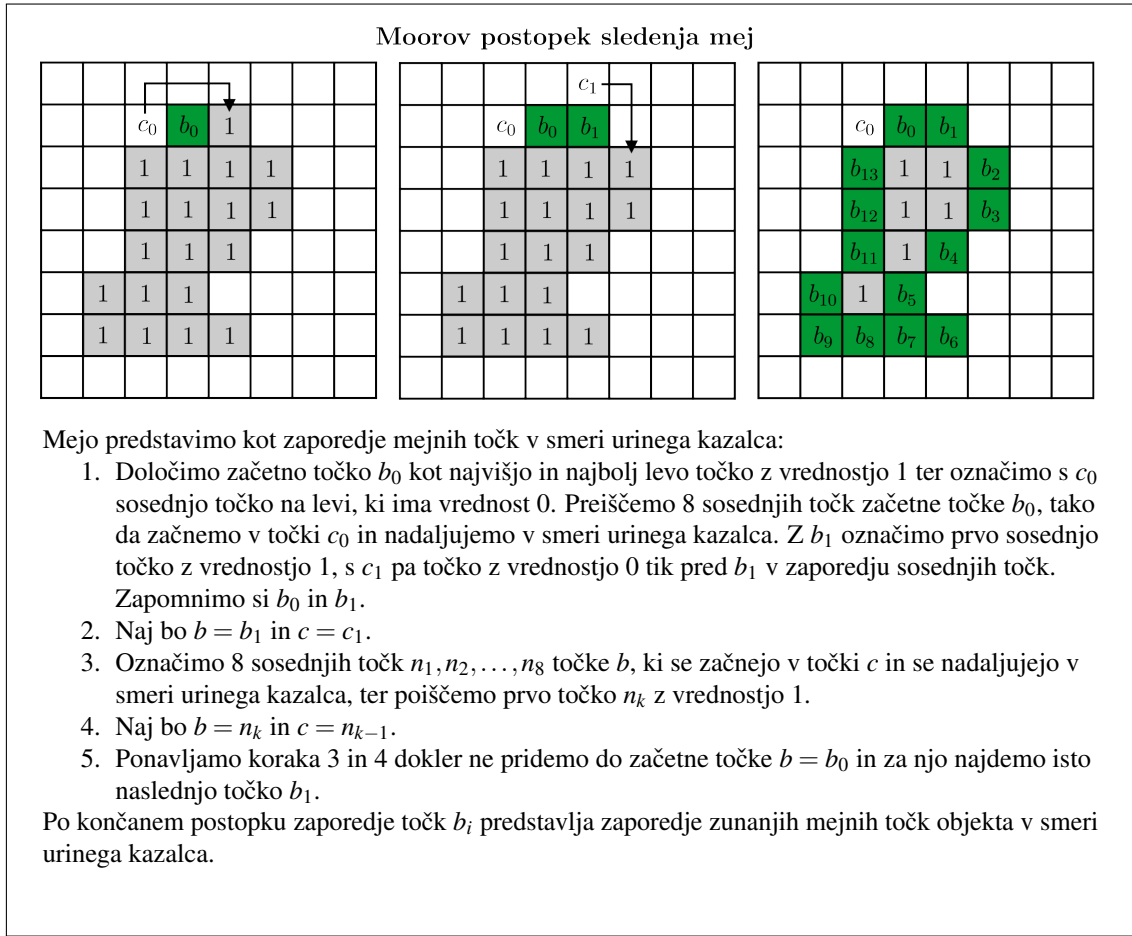
Najosnovnejša značilnica zunanje lastnosti področja je njegova površina A , ki je določena s številom slikovnih elementov področja. Meje področja v binarni sliki (0 – ozadje, 1 - objekt) lahko določimo s Moorovim postopkom sledenja mej (slika 13.2). Obseg P področja je enak dolžini njegove meje. Opis oblike področja, ki je neodvisen od velikosti in orientacije objekta je krožno razmerje:

$$K = \frac{4\pi A}{P^2} \quad (13.1)$$

ki ima vrednost 1 za področja v obliki kroga, sicer pa je manjše od 1.

Notranje lastnosti področja

Notranje lastnosti področja najbolj preprosto opišemo s statističnimi momenti sivinskih vrednosti področij slike. Naj bo $p(z)$ verjetnostna porazdelitev sivinskih vrednosti znotraj posameznih



Slika 13.2: Moorov postopek za sledenje in določanje meje objekta.

področij, ki jo ocenimo z normalizacijo histograma sivinskih vrednosti znotraj posameznih področij slike. Centralni moment n -te stopnje izračunamo kot:

$$\mu_n = \sum_{i=0}^{L-1} (z_i - m)^n p(z_i), \text{ kjer povprečje } m \text{ izračunamo kot } m = \sum_{i=0}^{L-1} z_i p(z_i) \quad (13.2)$$

Ničti moment μ_0 je vedno enak 1, prvi moment pa 0. Drugi moment predstavlja varianco σ^2 , ki je značilnica razpršenosti oz. kontrasta sivinskih vrednosti, in ga lahko uporabimo za opis gladkosti področja:

$$G = \frac{1}{1 + \sigma^2} \quad (13.3)$$

Ta ima vrednost 1 na popolnoma homogenih področjih in se približuje 0 na področjih z veliko varianco.

Teksturo objekta lahko opišemo tudi z osnovnimi komponentami, naprimer tako, da izračunamo gradient slike $\nabla I = [g_x \ g_y]^T$ in za področje objekta sestavimo $N \times 2$ matriko gradientov ∇I ter določimo kovariančno matriko gradientov $C = (\nabla I)^T \nabla I$ z dimenzijami 2×2 . Nato določimo lastni vrednosti matrike $C \rightarrow \lambda_1, \lambda_2$, ki kodirata lastnosti teksture na objektu. Za objekte, kjer je tekstura izrazito orientirana v določeno smer, lahko izpeljemo značilnico:

$$R = 1 - \frac{\min\{\lambda_1, \lambda_2\}}{\max\{\lambda_1, \lambda_2\}}. \quad (13.4)$$

Značilnica R bo imela vrednost 0, če bo teksura orientirana enakomerno v vse smeri, oziroma vrednost 1, če bo izrazito orientirana v eno smer.

13.1.3 Razvrščanje

Razvrščanje (klasifikacija) je ključni korak v vizualni kontroli kakovosti, saj omogoča avtomatsko odločanje o ustreznosti ali neustreznosti objektov na podlagi njihovih značilnic. Obstajajo različne metode razvrščanja, od enostavnih pragovnih metod do naprednih algoritmov strojnega učenja. Spodaj so opisane nekatere izmed najpogostejše uporabljenih metod in njihove implementacije v Python knjižnicah.

- Pragovna metoda (ang. *thresholding*): Pragovna metoda je ena najpreprostejših tehnik razvrščanja, kjer vrednost določene značilnice (npr. povprečna intenziteta sivinske slike) primerjamo s prednastavljeno mejno vrednostjo. Če je vrednost nad pragom, je objekt razvrščen v eno kategorijo, sicer v drugo.
- K-najbližjih sosedov (KNN; ang. *K-nearest neighbor*): KNN je preprost nadzorovan učni algoritem, ki nove vzorce razvršča glede na razdaljo do k najbližjih že znanih primerov v učnem sklopu. Primer uporabe:

```
1 from sklearn.neighbors import KNeighborsClassifier
2 import numpy as np
3
4 # Učni podatki (primeri značilnic)
5 X_train = np.array([[1, 2], [2, 3], [3, 1], [5, 5], [6, 7]])
6 y_train = np.array([0, 0, 0, 1, 1]) # Razredi
7
8 # Novi podatki za razvrščanje
9 X_test = np.array([[4, 4]])
10
11 # Ustvarimo in naučimo model
12 knn = KNeighborsClassifier(n_neighbors=3)
13 knn.fit(X_train, y_train)
14
15 # Napoved
16 prediction = knn.predict(X_test)
17 print("Razred:", prediction)
```

- Naključni gozd (RF; ang. *Random Forest*): Naključni gozd je ansambelski model, ki uporablja več odločilnih dreves in glasovanje za končno napoved. Primer uporabe:

```
1 from sklearn.ensemble import RandomForestClassifier
2
3 # Ustvarimo model
4 rf = RandomForestClassifier(n_estimators=100, random_state=42)
5
6 # Učenje modela
7 rf.fit(X_train, y_train)
8
9 # Napoved
10 prediction = rf.predict(X_test)
11 print("Razred:", prediction)
```


- XGBoost (ang. *Extreme Gradient Boosting*): XGBoost je optimizirana različica gradientnega ojačevanja, ki omogoča hitro in učinkovito razvrščanje. Primer uporabe:

```

1 import xgboost as xgb
2
3 # Ustvarimo model in ga naučimo
4 xgb_model = xgb.XGBClassifier(use_label_encoder=False,
5                               eval_metric="logloss")
6 xgb_model.fit(X_train, y_train)
7
8 # Napoved
9 prediction = xgb_model.predict(X_test)
10 print("Razred:", prediction)

```

- Metoda podpornih vektorjev (SVC; ang. *Support Vector Machine*): SVC je model, ki išče optimalno mejo za ločevanje razredov v večdimenzionalnem prostoru, pri čemer optimalna meja temelji na identifikaciji vzorcev ob meji tako, da so ti vzorci (rečemo jim tudi podporni vektorji) hkrati čimbolj oddaljeni od optimalne meje. Primer uporabe:

```

1 from sklearn.svm import SVC
2
3 # Ustvarimo model
4 svc = SVC(kernel="linear")
5
6 # Učenje modela
7 svc.fit(X_train, y_train)
8
9 # Napoved
10 prediction = svc.predict(X_test)
11 print("Razred:", prediction)

```

Vsaka od zgornjih metod ima svoje prednosti in slabosti. Pragovne metode so enostavne, vendar ne fleksibilne. KNN je enostaven, a počasen pri velikih podatkovnih zbirkah. Random Forest in XGBoost sta robustna modela, primerna za kompleksne podatke, medtem ko SVC deluje dobro pri dobro ločljivih podatkih. Izbira metode je odvisna od značilnic vhodnih podatkov in zahtev aplikacije.

13.2 Vrednotenje sistemov odločanja

Pri ocenjevanju učinkovitosti sistemov odločanja uporabljamo različne metrike, ki nam pomagajo razumeti, kako dobro model ločuje med pozitivnimi in negativnimi primeri. Nekatere ključne metrike vključujejo občutljivost (senzitivnost), specifičnost, ter ROC krivuljo in AUC metriko.

13.2.1 Občutljivost in specifičnost

Občutljivost (ang. *sensitivity* ali *recall*) meri, koliko dejanskih pozitivnih primerov je pravilno prepoznanih kot pozitivni:

$$\text{Občutljivost} = \frac{TP}{TP + FN} \quad (13.5)$$

kjer je:

- TP (ang. *True Positive*) – pravilno prepoznani pozitivni primeri,

- FN (ang. *False Negative*) – napačno prepoznani negativni primeri.

Specifičnost (ang. *specificity*) meri, koliko dejanskih negativnih primerov je pravilno označenih kot negativni:

$$\text{Specifičnost} = \frac{\text{TN}}{\text{TN} + \text{FP}} \quad (13.6)$$

kjer je:

- TN (ang. *True Negative*) – pravilno prepoznani negativni primeri,
- FP (ang. *False Positive*) – napačno prepoznani pozitivni primeri.

13.2.2 ROC krivulja in AUC metrika

ROC (ang. *Receiver Operating Characteristic*) krivulja prikazuje razmerje med občutljivostjo in nespecifičnostjo (tj. *1 - specifičnost*) pri različnih pragovnih vrednostih. Krivulja nam omogoča vizualno analizo delovanja razvrščevalnika.

AUC (ang. *Area Under the Curve*) metrika meri površino pod ROC krivuljo in predstavlja splošno zmogljivost modela:

$$0,5 \leq \text{AUC} \leq 1 \quad (13.7)$$

kjer vrednost 1 pomeni popolno ločevanje pozitivnih in negativnih primerov, vrednost 0.5 pa pomeni naključno napovedovanje.

ROC krivuljo in AUC vrednost lahko izrišemo in izračunamo v Pythonu s knjižnico sklearn:

```
1 from sklearn.metrics import roc_curve, auc
2 import matplotlib.pyplot as plt
3
4 # Primer podatkov: resnične oznake in napovedane verjetnosti
5 y_true = [0, 0, 1, 1]
6 y_scores = [0.1, 0.4, 0.35, 0.8]
7
8 # Izračun ROC krivulje
9 fpr, tpr, _ = roc_curve(y_true, y_scores)
10 roc_auc = auc(fpr, tpr)
11
12 # Prikaz ROC krivulje
13 plt.figure()
14 plt.plot(fpr, tpr, color="darkorange", lw=2, label="ROC curve (area = %0.2f)" %
15         roc_auc)
16 plt.plot([0, 1], [0, 1], color="navy", lw=2, linestyle="--")
17 plt.xlabel("1 - Specifičnost")
18 plt.ylabel("Občutljivost")
19 plt.title("ROC krivulja")
20 plt.legend(loc="lower right")
21 plt.show()
```

Te metrike so ključne za vrednotenje sistemov odločanja, še posebej na področjih, kot so medicinska diagnostika, varnostni sistemi in strojno učenje.

13.3 Vaje z rešitvami

Koda vaje je dostopna na [Git repozitoriju](#). **Gradivo** za vajo vsebuje barvni sliki osvetlitev-ozadje.jpg in osvetlitev-ospredje.jpg, ki sta bili zajeti z osvetlitvijo objektov zanimanja (farmacevtskih

tablet) iz ozadja in ospredja. Na slikah so farmacevtske tablete različnih oblik, barv, tekstur, nekatere tablete pa imajo tudi zarezo.

Najprej uvozimo osnovne Python knjižnice:

```
1 import PIL.Image as im
2 import scipy.ndimage as ni
3 import matplotlib.pyplot as plt
4 import matplotlib.cm as cm
5 import numpy as np
6 import os.path
7 import sys
8
9 from rvlb import imageGradient, loadImage, showImage, toGray
```

Vaja 13.1 Naložite sivinsko sliko osvetlitev-ozadje.jpg in jo z upragovljanjem pretvorite v binarno sliko tako, da določite optimalen prag t , tj. sivinsko vrednost, ki področje tablet loči od ozadja. Optimalni prag t določite s pomočjo histograma sivinske slike. ■

Za izračun histograma si pripravimo dve funkciji. Prva je funkcija `getIndices()`, ki naj pretvori vrednosti sivinskih slikovnih elementov (pikslov) v indekse histograma. Deluje naj tako, da normalizira vhodne vrednosti glede na dano območje (`iMinVal`, `iMaxVal`) in jih preslika na število predalčkov (`iBins`). Normalizacija poteka po formuli:

$$\text{idx} = \text{round} \left(\frac{iData - iMinVal}{iMaxVal - iMinVal} \times (iBins - 1) \right),$$

pri čemer se rezultat zaokroži na cela števila in pretvori v 32-bitni nepredznačeni celoštevilski tip (`uint32`). Primer uporabe: če je `iData = [10, 20, 30]`, `iBins = 4`, `iMinVal = 0` in `iMaxVal = 40`, potem je izračun:

$$\text{idx} = \text{round} \left(\frac{[10, 20, 30]}{40} \times 3 \right) = [1, 2, 2].$$

Druga funkcija, `hist1D(iData, iBins, iMinVal=None, iMaxVal=None)`, naj izračuna 1D histogram sivinskih vrednosti slike z uporabo funkcije `getIndices`. Če območje vrednosti (`iMinVal` in `iMaxVal`) ni podano, se avtomatsko določita kot minimum in maksimum vhodnih podatkov, pri čemer se pri `iMaxVal` doda majhna vrednost ($1e-7$), da se zagotovi, da je večji od največje vrednosti. Nato se vrednosti pretvorijo v indekse, inicializira se ničelno polje velikosti `iBins`, in z iteracijo po indeksih se ustrezni predalčki inkrementirajo. Izhod je polje `histData`, kjer vsak element predstavlja število pikslov v ustreznem predalčku.

Rešitev zapišemo:

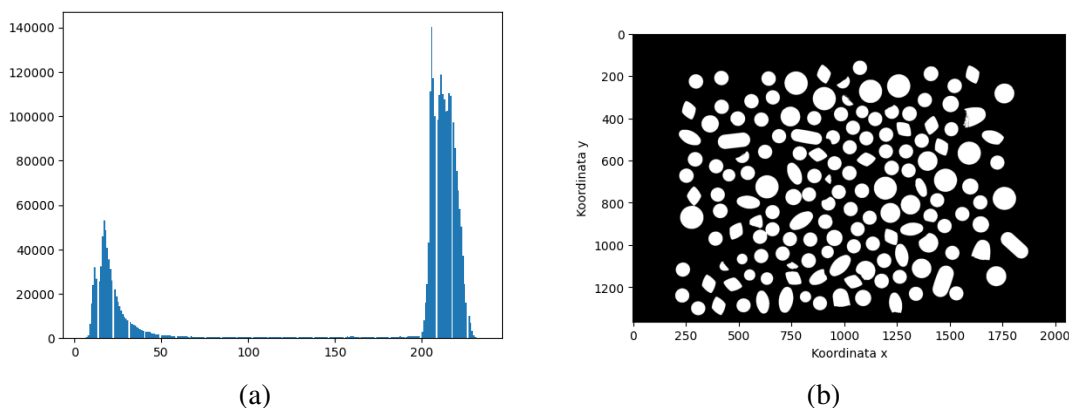
```
1 # funkcija za pretvorbo sivinskih vrednosti v indekse
2 def getIndices( iData, iBins, iMinVal, iMaxVal ):
3     # pretvori v indeks polja
4     idx = np.round( (iData - iMinVal) / (iMaxVal - iMinVal) * (iBins-1) )
5     # vrni indekse
6     return idx.astype('uint32')
7
8 # funkcija za izracun 1D histograma
9 def hist1D( iData, iBins, iMinVal=None, iMaxVal=None ):
10    # doloci obmocje sivinskih vrednosti
11    if iMinVal==None: iMinVal = np.min(iData)
```

```

12     if iMaxVal==None: iMaxVal = np.max(iData)+1e-7
13     # pretvorba sivinskih vrednosti v indekse
14     idx = getIndices( iData, iBins, iMinVal, iMaxVal )
15     # izracunaj histogram
16     histData = np.zeros((iBins,))
17     for i in idx:
18         histData[i] += 1.0
19     # vrni histogram
20     return histData
21
22 # nalaganje slik
23 imgB = loadImage('osvetlitev-ozadje.jpg')
24 imgF = loadImage('osvetlitev-ospredje.jpg')
25
26 # prikaži histogram indoloči optimalen prag
27 imgBG = toGray( imgB )
28 plt.hist( imgBG.flatten(), 256 )
29 iThr = 100 # izbran prag
30 # prikaz upravljenja slike z osvetlivijo iz ozadja
31 imgT = 255 * (imgBG < iThr).astype('uint8')
32 showImage( imgT )

```

Testna koda demonstrira analizo histograma sivinskih vrednosti dveh slik (ozadje in ospredje) za določitev optimalnega praga ($iThr$). Sliki se naložita s funkcijo `loadImage`, ozadje se pretvori v sivinsko sliko z `toGray`, histogram se prikaže z `plt.hist` (256 predalčkov), prag ($iThr$, npr. 100) se ročno nastavi, in ustvari se binarna slika, kjer so piksli pod pragom beli (255), ostali pa črni (0). Končni rezultat se prikaže s `showImage`. Skupaj funkciji `getIndices` in `hist1D` omogočata analizo porazdelitve sivinskih vrednosti, kar je ključno za segmentacijo slike, na primer ločevanje ozadja in ospredja. Rezultat ločevanja tablet od ozadja je prikazan na sliki 13.3.



Slika 13.3: Prikaz (a) histograma slike `osvetlitev-ozadje.jpg` in (b) binarna slika ustvarjena s pragom $iThr=100$.

Vaja 13.2 Na osnovi upravljenega slike ozadja s preizkusite vpliv funkcij za morfološko filtriranje `erosion()`, `dilation()`, `opening()` in `closing()` v Python knjižnici `skimage.morphology`. Jedro za filtriranje določite s funkcijo `disk()`.

Po filtriranju lahko vsakemu objektu pripišete unikatno oznako s funkcijo `label()` v knjižnici `skimage.measure`. Koliko objektov oz. tablet je na sliki? Preverite vpliv morfološkega

filtriranja na število zaznanih tablet v sliki. ■

Morfološke operacije nad binarno sliko 13.3b izvedemo kot:

```
1 from skimage.morphology import erosion, dilation, opening, closing
2 from skimage.morphology import disk
3 from skimage.measure import label
4 # poišci vse objekte na sliki
5
6 # erozija
7 imgE = erosion(imgT, disk(3)) #3-> velikost strukturnega elementa
8 showImage(imgE, 'Erozija')
9 # dilacija
10 imgD = dilation(imgT, disk(3))
11 showImage(imgD, 'Dilacija')
12 # erozija-dilacija
13 imgO = opening(imgT, disk(3))
14 showImage(imgO, 'Odpiranje (erozija+dilacija)')
15 # dilacija-erozija
16 imgC = closing(imgT, disk(3))
17 showImage(imgC, 'Zapiranje (dilacija+erozija)')
18 # označevanje objektov
19 iLabel = label( imgO, neighbors=8 )
20 numL = np.max( iLabel )
21 msg = 'Najdeno stevilo objektov je %d.' % numL
22 # prikaži sliko objektov
23 showImage( iLabel.astype('uint8') )
```

Rezultat prikazuje slika 13.4. Za nadaljnjo uporabo v analizi objektov izberemo sliko 13.4c, pridobljeno z operacijo odpiranja.

Nato izvedemo še označevanje objektov s funkcijo `label()`:

```
1 # označevanje objektov
2 iLabel = label( imgO, connectivity=2 )
3 numL = np.max( iLabel )
4 # prikaži sliko objektov
5 showImage( iLabel.astype('uint8'), 'Označeni objekti' )
6 print(f'Najdeno stevilo objektov je {numL}.')
```

Rezultat prikazuje slika 13.5, testna koda pa vrne naslednji izpis:

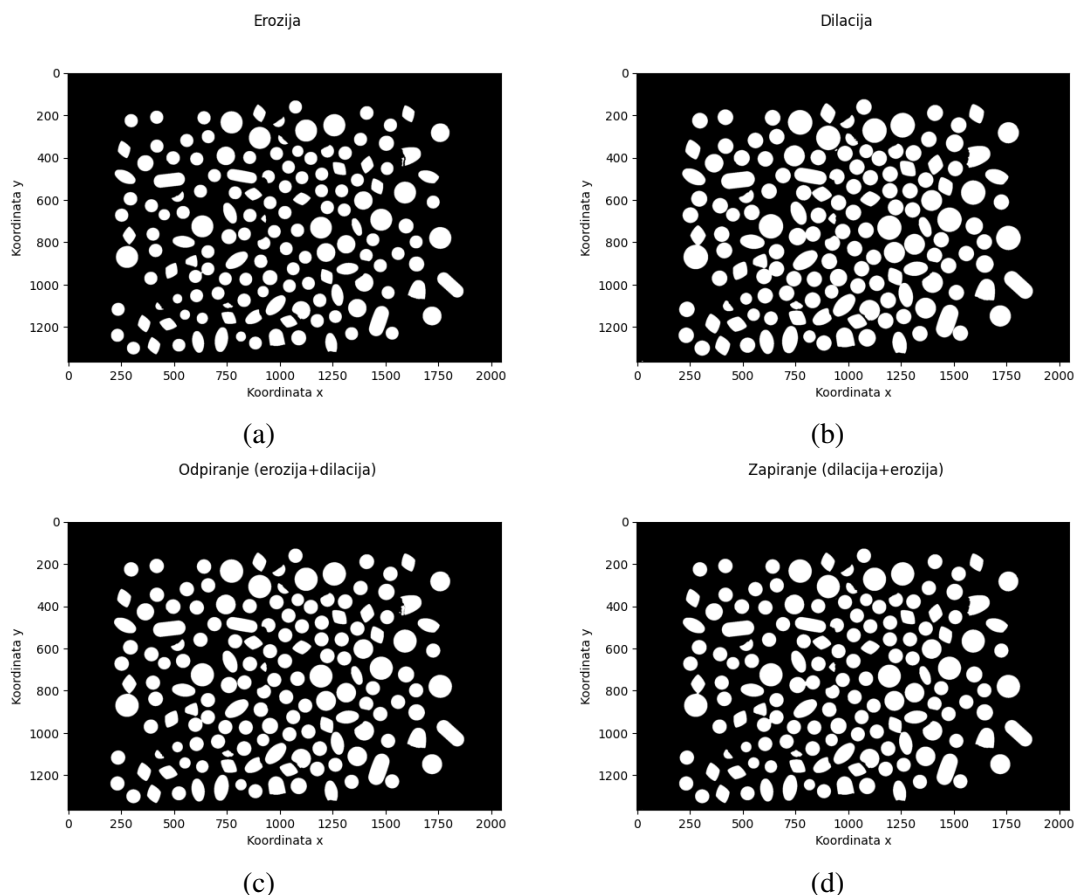
```
Najdeno stevilo objektov je 152.
```

Vaja 13.3 Napišite funkcijo, ki določi koordinate meje posameznega področja na 2D sliki:

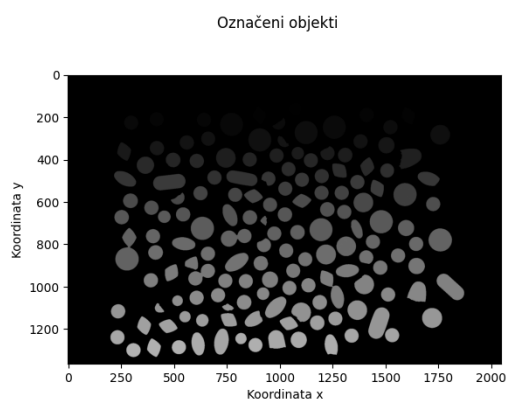
```
1 def trackContour( iLabel, iObject ):
2     # Tu napišite python kodo funkcije
3     return oContour
```

kjer je `iLabel` slika označenih objektov, `iObject` pa je oznaka objekta, katerega mejo želimo določiti. Funkcija vrne spremenljivko `oContour` v obliki matrike $2 \times M$ z zaporedjem koordinat meje $[x_i y_i]^T, i = 1, \dots, M$.

Določite meje vseh označenih objektov iz vaje 13.2. Prikažite sliko `osvetlitev-ospredje.jpg`



Slika 13.4: Morfološke operacije nad binarno sliko 13.3b: (a) erozija, (b) dilacija, (c) odpiranje in (d) zapiranje.



Slika 13.5: Slika označenih objektov, kjer posamezna sivina predstavlja oznako objekta. Skupaj je bilo v sliki najdenih 152 objektov.

ter vanjo vrišite meje označenih objektov kot je prikazano na sliki 13.6 *levo*. ■

Funkcija `trackContour` sledi konturi objekta v binarni maski z uporabo algoritma sledenja robu (ang. *boundary tracking*). Najprej preveri velikost objekta in zavrne objekte z manj kot 10 piksli. Nato poišče začetno točko konture kot zgornjo levo točko objekta in iterativno sledi robu s pomožno funkcijo `findNextPoint`, ki preverja 8-sosednjih pikslov v nasprotni smeri urinega

kazalca, začeni od prejšnjega odmika (`iOffset`). Vsaka najdena točka se doda v izhodno polje `oContour`, dokler se ne zapre zanka (ko naslednja točka ustreza prvi točki konture). Funkcija `findNextPoint` za iskanje naslednje točke konture na podlagi trenutne pozicije in prejšnjega odmika, kar zagotavlja dosledno sledenje robu objekta. Rezultat je 2D polje točk konture v obliki polja $2 \times N$, kjer N predstavlja število točk na konturi. Rešitev zapišemo:

```

1 def findNextPoint( iObjectMask, iCurPoint, iOffset ):
2     """Funkcija za iskanje naslednje točke"""
3     # preberi velikost slike
4     dy, dx = iObjectMask.shape
5     # preberi trenutno položaj
6     x, y = iCurPoint
7     # definiraj zaporedje odmikov
8     lOffsets = np.array( ( (-1,-1,0,1,1,1,0,-1), (0,-1,-1,-1,0,1,1,1) ) )
9     sx = lOffsets[0,:]; sy = lOffsets[1,:]
10    lOffsetIdx = np.where( np.sum( np.abs( lOffsets - iOffset ), axis=0 )
11                          == 0 )[0]
12    for i in range(len(sx)):
13        j = (i + lOffsetIdx) % len(sx)
14        if (x+sx[j])<dx and (x+sx[j])>=0 and \
15            (y+sy[j])<dy and (y+sy[j])>=0:
16            if iObjectMask[y+sy[j], x+sx[j]]:
17                # določi koordinate naslednje točke konture
18                oNextPoint = np.array( (x+sx[j], y+sy[j]) )
19                # določi koordinate prejšnje točke zunaj konture
20                oPrevPoint = np.array( (x+sx[j-1], y+sy[j-1]) )
21                return oNextPoint, oPrevPoint-oNextPoint
22    return iCurPoint, iOffset
23
24 def trackContour( iLabel, iObject ):
25     """Funkcija za določanje konture objekta"""
26     # definiraj izhodno spremenljivko
27     oContour = np.array( [] )
28     # preveri velikost objekta in preskoči objekte
29     # po velikosti manjše od 10 slikovnih elementov
30     iObjectMask = iLabel == iObject
31     y, x = np.where( iObjectMask )
32     if x.size < 10:
33         print(f'Objekt {iObject} ima velikost {x.size} < 10')
34         return oContour
35     # poišči zaceto točko na konturi kot zgornjo-levo točko
36     idy = np.where( y == np.min(y) )[0]
37     idx = np.where( x[idy] == np.min(x[idy]) )[0]
38     oContour = np.array( (x[idy[idx]], y[idy[idx]]) ).reshape( (2,1) )
39     # iterativno išči točke na konturi
40     iOffset = np.array((-1,0)).reshape((2,1))
41     while True:
42         # trenutna točka
43         oCurPoint = oContour[:,-1]
44         # naslednja točka
45         oNextPoint, iOffset = findNextPoint( iObjectMask, oCurPoint, iOffset )
46         # preveri ali je enaka prejšnji ali prvi točki
47         if np.all( oCurPoint == oNextPoint ) or \
48            np.all( oNextPoint.flatten() == oContour[:,0] ):
49             # da, prekini zanko
50             break

```

```

50     # dodaj točko na konturo
51     oContour = np.hstack( (oContour, oNextPoint) )
52     # vrni konturo
53     return oContour

```

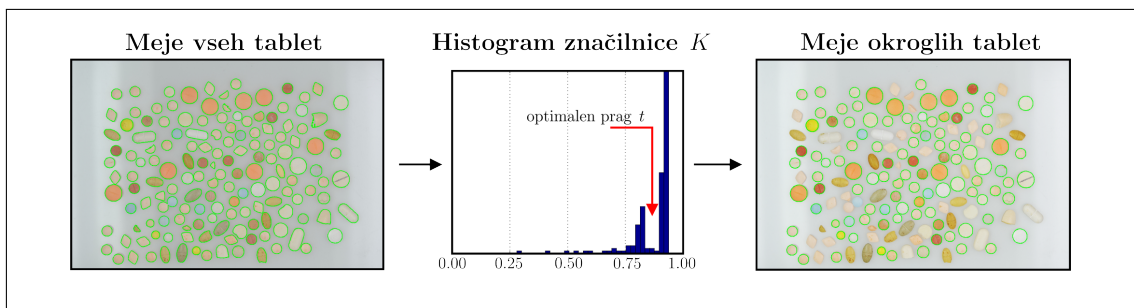
S funkcijo `trackContour()` na sliki označenih objektov 13.5 določimo konturo vsakemu objektu posebej in konture nato narišemo v sliko kot:

```

1 def plotContour( oContour ):
2     """Prikaz konture na sliki"""
3     plt.plot(oContour[0,:], oContour[1,:], '-g')
4     plt.plot(oContour[0,(-1,0)], oContour[1,(-1,0)], '-g')
5
6 oContours = []
7 for iObject in range(0,numL):
8     x,y = np.where( iLabel == (iObject+1) )
9     # preskoči objekte po velikosti manjše od 10 slikovnih elementov
10    if x.size > 10:
11        # z uporabo Moorovega algoritma
12        oContours.append( trackContour( iLabel, iObject+1 ) )
13
14 # prikaži barvno sliko z osvetlitvijo iz ospredja in obriši konture
15 showImage( imgF )
16 for i in range(len(oContours)):
17     plotContour( oContours[i] )

```

Rezultat prikazuje slika 13.6 levo, kjer so konture objektov označene z zeleno barvo.



Slika 13.6: Prikaz mej posameznih objektov in razvrščanje z upravljanjem značilnice K (enačba (13.1)).

Vaja 13.4 Napišite funkcijo, ki izloči značilnice označenih objektov na 2D sliki:

```

1 def extractFeatures( iImage, iLabel ):
2     # Tu napišite python kodo funkcije
3     return oFeatures

```

kjer je `iImage` sivinska slika z osvetlitvijo iz ospredja, `iLabel` slika označenih objektov, katerih značilnice želimo izračunati. Funkcija vrne spremenljivko `oFeatures` v obliki vektorja z značilnicami $[A, P, K, \mu_2, G, R]^T$, ki ima dimenzije $6 \times N_{obj}$. Značilnice notranjih lastnosti objektov določite na erodirani maski objektov, ki jo dobite z ukazom `iMask = erosion(iLabel, disk(3))`.

Narišite 1D histograme posameznih značilnic za vse objekte v sliki. Razmislite katera oz. katere od značilnic bi bile primerne za zaznavo okroglih tablet in tablet z zarezo? ■

Funkcija `extractFeaturesSingle()` izračuna geometrijske in teksturne značilnice objekta na podlagi vhodne slike (`iImage`) in binarne maske (`iMask`). Najprej določi konturo objekta s funkcijo `trackContour()` in izračuna osnovne geometrijske značilnice: površino objekta (A), obseg konture (P) ter krožnost ($K = 4\pi A/P^2$). Nato uporabi morfološko erozijo za odstranitev robov objekta in izračuna teksturne značilnice iz sivinske slike: normaliziran histogram (p), srednjo vrednost (m), varianco (μ_2), mero homogenosti ($G = 1/(1 + \mu_2)$) ter mero strukture gradientov (R), ki temelji na lastnih vrednostih kovariančne matrike gradientov slike. Vse značilnice združi v izhodni vektor `oFeatures` v obliki (A, P, K, μ_2, G, R) , ki opisuje lastnosti objekta. Funkcija naj zavrne premajhne objekte (manj kot 10 pikselov) z vrnitvijo `None`. Rešitev zapišemo kot:

```
1 from skimage.morphology import erosion
2 from skimage.morphology import disk
3
4 def extractFeaturesSingle( iImage, iMask ):
5     """Funkcija za določanje značilnic objekta"""
6     # izračun dolžine konture
7     def lengthContour( oContour ):
8         dC = np.diff( np.hstack( (oContour, oContour[:,0].reshape((2,1))) ),
9             axis=-1 )
10        return np.sum( np.sqrt(np.sum(dC**2.0,axis=0)) )
11    # določi konturo objekta
12    iMask = iMask.astype('uint8')
13    oContour = trackContour( iMask, 1 )
14    # določi zunanje značilnice področja
15    A = np.count_nonzero( iMask )
16    if A < 10:
17        print(f'Objekt {iObject} ima velikost {A} < 10')
18        return None
19    P = lengthContour( oContour )
20    K = 4.0 * np.pi * A / (P**2.0)
21    # odstrani sivinsko informacijo na robu z erozijo
22    iMask = erosion( iMask, disk(3) )
23    # izlušči informacijo
24    iImageG = toGray( iImage )
25    oGx, oGy = imageGradient( iImageG )
26    y, x = np.where( iMask>0 )
27    # določi notranje značilnice področja
28    p = hist1D( iImageG[y, x].flatten(), iBins=256, iMinVal=0, iMaxVal=255 )
29    p = p / ( np.sum( p ) + 1e-7 )
30    m = np.sum( np.array( range(256) ) * p )
31    mu2 = np.sum( np.array( range(256) - m )**2.0 * p )
32    G = 1.0 / ( 1.0 + mu2 )
33    Gxy = np.vstack( ( oGx[y, x].flatten(), oGy[y, x].flatten() ) )
34    Eigvals = np.linalg.eigvals( np.dot( Gxy, Gxy.transpose() ) )
35    R = 1 - np.min( Eigvals ) / np.max( Eigvals )
36    # sestavi vektor značilnic in ga vrni
37    oFeatures = np.array( (A,P,K,mu2,G,R) )
38    return oFeatures
39
40 # preizkus funkcije
41 iObject = 3 # oznaka objekta
42 oFeatures = extractFeaturesSingle( imgF, iLabel==(iObject+1) ) # izračun
    značilnic
```

```
42 | print('features =', oFeatures)
```

Vektor značilnic za objekt z oznako `iObject = 3` v sliki 13.5 je:

```
features = [3.81800000e+03 2.27965512e+02 9.23224857e-01 3.94994389e+01
           ↪ 2.46917001e-02 2.21304642e-01]
```

Funkcijo `extractFeaturesSingle()` uporabimo za izračun značilnic na posameznih objektih in potem združimo rezultat v matriko dimenzij $M \times 6$, kjer je M število objektov, vsak s šestimi značilnicami. Rešitev zapišemo:

```
1 | def extractFeatures( iImage, iLabel ):
2 |     """Funkcija za določanje značilnic vseh objektov v sliki"""
3 |     # pripravi vhodne podatke
4 |     iImage = np.asarray( iImage )
5 |     iLabel = np.asarray( iLabel )
6 |     numL = np.max( iLabel )
7 |     # določi značilnice za vsak objekt
8 |     oFeatures = np.array( () )
9 |     for iObject in range(numL):
10 |         # izpis poteka izračuna značilnic
11 |         if iObject==0:
12 |             oFeatures = extractFeaturesSingle( iImage, iLabel == (iObject+1)
13 |                                                 ).reshape( (1,6) )
14 |         else:
15 |             oFeaturesSingle = extractFeaturesSingle( iImage, iLabel ==
16 |                                                       (iObject+1) )
17 |             if oFeaturesSingle is not None:
18 |                 oFeaturesSingle = oFeaturesSingle.reshape( (1,6) )
19 |                 oFeatures = np.vstack( (oFeatures, oFeaturesSingle))
20 |     # končaj in vrni značilnice
21 |     return oFeatures
```

Rezultat prikazuje slika 13.7. Za razvrščanje oz. zaznavanje okroglih tablet je primerna značilnica kompaktnosti oblike K , medtem ko je za zaznavanje tablet z zarezo primerna značilnica izrazitosti orientacije R .

Vaja 13.5 Na osnovi slik označenih objektov `iLabel` in značilnic $[A, P, K, \mu_2, G, R]^T$ teh objektov načrtajte binarne razvrševalnike, ki bodo odgovorili na naslednja vprašanja:

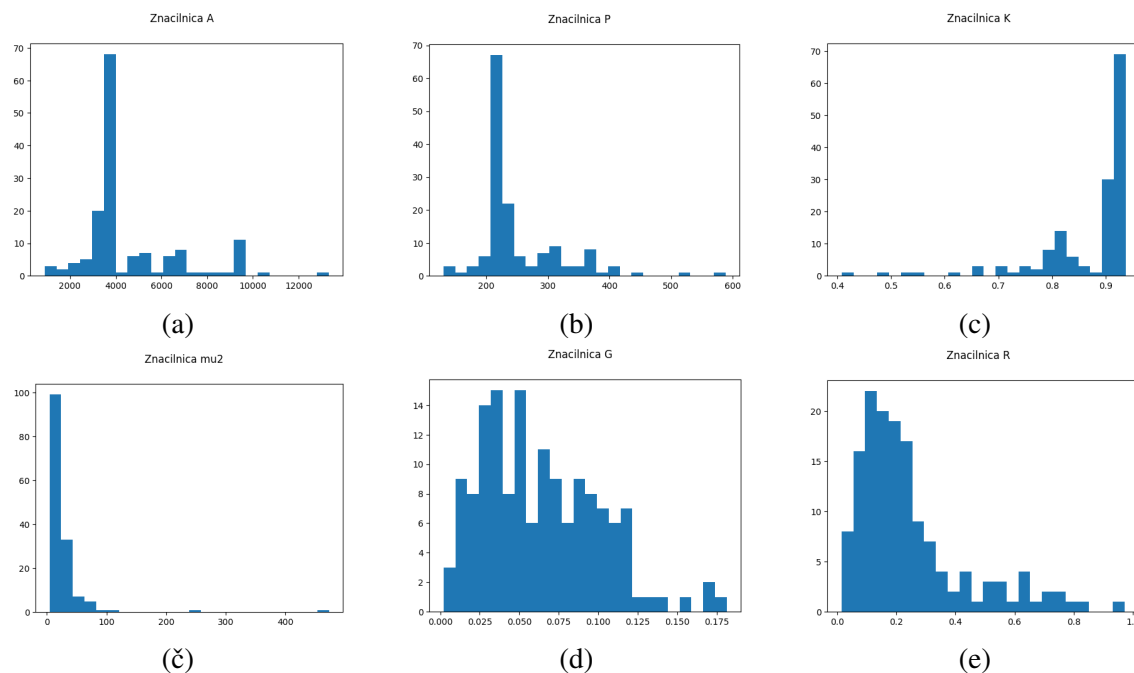
- Koliko tablet je na sliki?
- Koliko tablet je okroglih tablet?
- Koliko tablet je okroglih tablet z zarezo?

Razmislite na kakšen način bi vrednotili zanesljivost delovanja tega procesa razvrščanja. ■

Za odgovor na vprašanja bomo na podlagi opazovanja porazdelitev in pomena značilnic v sliki 13.7 ustvarili enostavne (hierarčne) binarne odločitvene funkcije in tako odgovorili na postavljena vprašanja.

Koliko tablet je na sliki? Privzamemo, da je pri trenutni postavitvi posamezna tableta velika vsaj 500 piklov po površini (značilnica A), kar dosežemo z upravljanjem po tej značilnici:

```
1 | idxTablete = np.where( (oFeatures[:,0]>500) )[0]
2 | #idxTablete = np.where( (oFeatures[:,0]>500) * (oFeatures[:,0]<12e+3) )[0]
3 | print('fNa sliki je {idxTablete.size} tablet.')
4 | # prikažemo barvno sliko z označenimi tabletami
```



Slika 13.7: Histogrami značilnic vseh objektov v sliki 13.5: (a) površina objekta A, (b) obseg objekta P, (c) kompaktnost objekta K, (č) varianca intenzitete $\mu_2 = \sigma^2$, (d) gladkost sivinskih vrednosti G in (e) izrazitost orientacije R. Opisi značilnic so podani v podpoglavju 13.1.2.

```

5 plt.imshow(imgF)
6 plt.suptitle('Slika tablet')
7 plt.xlabel('Koordinata x')
8 plt.ylabel('Koordinata y')
9 for i in range(idxTablete.size):
10     plotContour( oContours[idxTablete[i]] )
11 plt.show()

```

Objekti, ki ustrezajo imajo označene konture v sliki 13.8a. Dobimo odgovor:

Na sliki je 148 tablet.

Koliko tablet je okroglih tablet? Za vsak objekt, ki smo ga označili kot *tableta* nadalje preverimo še značilnico kompaktnosti oblike K . Prag postavimo na $K > 0,9$:

```

1 idxTableteOkrogle = np.where( (oFeatures[idxTablete,2]>0.9) )[0]
2 print(f'Na sliki je {idxTableteOkrogle.size} okroglih tablet.')
3 # prikažemo barvno sliko z označenimi tabletami
4 plt.imshow(imgF)
5 plt.suptitle('Slika okroglih tablet')
6 plt.xlabel('Koordinata x')
7 plt.ylabel('Koordinata y')
8 for i in range(idxTableteOkrogle.size):
9     plotContour( oContours[idxTablete[idxTableteOkrogle[i]]] )
10 plt.show()

```

Objekti, ki ustrezajo imajo označene konture v sliki 13.8b. Dobimo odgovor:

Na sliki je 99 okroglih tablet.

Koliko tablet je okroglih tablet z zarezo? Za vsak objekt, ki smo ga označili kot *okrogla tableta* nadalje preverimo še značilnico izrazitosti orientacije teksture R . Prag postavimo na $R > 0,4$:

```
1 idxTableteOkroggleZareza = np.where(
    (oFeatures[idxTablete[idxTableteOkroggle],5]>0.4) )[0]
2 print(f'Na sliki je {idxTableteOkroggleZareza.size} okroglih tablet z zarezo.')
3 # prikažemo barvno sliko z označenimi tabletami
4 plt.imshow(imgF)
5 plt.suptitle('Slika okroglih tablet')
6 plt.xlabel('Koordinata x')
7 plt.ylabel('Koordinata y')
8 for i in range(idxTableteOkroggleZareza.size):
9     plotContour(
        oContours[idxTablete[idxTableteOkroggle[idxTableteOkroggleZareza[i]]] ]
10 plt.show()
```

Objekti, ki ustrezajo imajo označene konture v sliki 13.8c. Dobimo odgovor:

Na sliki je 10 okroglih tablet z zarezo.

Zanesljivost razvrščanja lahko ovrednotimo tako, da preštejemo (za posamezno vprašanje oz. kategorijo) števimo pravilno zaznanih tablet (TP) in število tistih, ki smo jih napačno označili (ločeno lažno označene in neoznačene; FP in FN). Nato lahko ovrednotimo občutljivost, specifičnost in točnost razvrščanja kot velevajo enačbe v podpoglavju 13.2.

13.4 Naloge in vprašanja

Poročilo naloge pripravite v obliki Python skripte, ki naj izvede zahtevane izračune oz. klice funkcij in izriše pripadajoče slike oz. grafe. Vaša dolžnost je, da v skripti uvozite ustrezne Python knjižnice in dodate morebitne podporne skripte. Skripta se mora izvesti in poustvariti rezultate, kot se zahtevajo v nalogah. Na vprašanja odgovorite v obliki komentarja, naprimer

Naloga X.1: To je moj odgovor..

Naloga 13.1

Za namen vrednotenja označite vse objekte v sliki v naslednje razrede:

- okrogle / podolgovate tablete (pripadajoča značka razreda 1 in 2)
- tablete z zarezo / brez zareze (pripadajoča značka razreda 4 in 8)

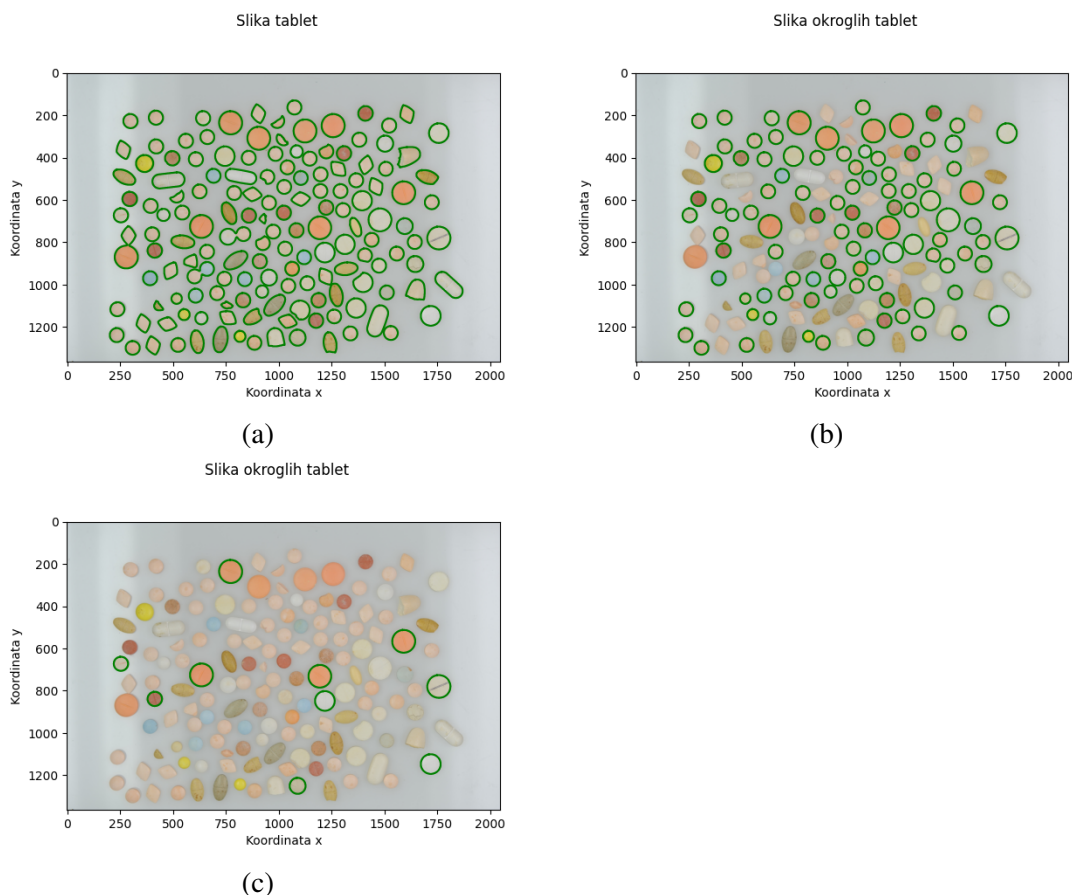
Značke razredov so določene tako, da jih lahko predstavimo z eno nepredznačeno 8-bitno sliko s po eno značko na bit (v tem primeru koristimo prve štiri bite). Ustvarite tako referenčno masko objektov.

Določite občutljivost in specifičnost razvrščanja objektov z razvrščevalnikom, ki smo ga razvili pri nalogi 5.

Naloga 13.2 Vrednotite kakovost razvrščanja z uporabo ROC² krivulja, pri čemer lahko uporabite implementacijo v Python knjižnici `sklearn`, in sicer funkcijo `sklearn.metrics.roc_curve`. Kakovost razvrščanja lahko opredelite z mero površine pod ROC krivuljo (AUC³) z uporabo funkcije `sklearn.metrics.auc`.

²ROC: ang. Receiver Operating Characteristics

³AUC: ang. Area Under the (ROC) Curve



Slika 13.8: Označeni objekti, ki odgovorijo na postavljeno vprašanje pri vaji 13.5: (a) objekt je tableta, (b) objekt je okrogla tableta in (c) objekt je okrogla tableta z zarezo.

Naloga 13.3 Uporabite referenčno masko objektov za učenje razvrščevalnika in vrednotite njegovo kakovost tako, da določite optimalno občutljivost in specifičnost ter AUC. Preizkusite naslednje razvrščevalnike:

- Naključna drevesa (RF^4): uporabite implementacijo `RandomForestClassifier` v knjižnici `sklearn.ensemble` (povezava na dokumentacijo)
- Podporne vektorje (SVM^5): uporabite implementacijo `svm` v knjižnici `sklearn` (povezava na dokumentacijo)
- K najbližjih sosedov ($k\text{-}NN^6$): uporabite implementacijo `NearestNeighbors` v knjižnici `sklearn.neighbors` (povezava na dokumentacijo)

Za vrednotenje razvrščevalnika je smiselno uporabiti križno validacijo, tj. učiti na delu podatkov in testirati na preostanku in nato še obratno. Delitev podatkov lahko naredite s funkcijo `train_test_split()` (povezava na dokumentacijo) v knjižnici `sklearn.model_selection`. Končne rezultate nato dobite s povprečenjem rezultatov pridobljenih na posameznih delih podatkov.

⁴RF: ang. *Random Forests*

⁵SVM: ang. *Support Vector Machines*

⁶k-NN: ang. *k Nearest Neighbors*

Dodatek

A Singularni razcep 215

- A.1 Matematična formulacija
- A.2 Povezava z lastnimi vrednostmi in lastnimi vektorji
- A.3 Lastnosti singularnega razcepa
- A.4 Geometrijska interpretacija
- A.5 Reševanje predoločenih sistemov enačb

B Aproksimacijska 3D toga preslikava . 219

- B.1 Toga preslikava v 3D
- B.2 Izpeljava aproksimacijske preslikave
- B.3 Aproksimacijska preslikava v računskih korakih

C Razvojna okolja 225

- C.1 Osnovna namestitve in konfiguracija Pythona
- C.2 Upravljanje Python razvojnih okolij s `pip`
- C.3 Upravljanje Python razvojnih okolij s `conda`
- C.4 Uporaba Docker virtualnega okolja
- C.5 Uporaba Docker mape
- C.6 Uporaba spletnih razvojnih okolij: Google Colab
- C.7 Lokalno razvojno okolje: Visual Studio Code

D Vodenje različic kode z Git 245

- D.1 Kaj je Git in zakaj ga uporabljamo?
- D.2 Namestitve Gita
- D.3 Osnovne nastavitve
- D.4 Ustvarjanje in kloniranje repozitorija
- D.5 Osnovni Git ukazi
- D.6 Uporaba vejitev
- D.7 Povezava z oddaljenim repozitorijem
- D.8 Pogoste napake in koristni namigi
- D.9 Zaključek
- D.10 Dobra praksa priprave Git repozitorija
- D.11 Zaključek

Bibliografija 255

- Knjige
- Članki
- Konferenčni članki
- Razno

O avtorju 259

A. Singularni razcep

Singularni razcep (SVD; ang. *Singular value decomposition*) [8, str. 73–74] je ena temeljnih matrikovnih faktorizacij v linearni algebri. Ta faktorizacija je vsestransko uporabno orodje v numerični linearni algebri, za analizo realnih ali kompleksnih matrik pri številnih numeričnih metodah v statistiki in obdelavi signalov in slik. Na primer za reševanje predoločenih sistemov enačb, za stiskanje podatkov in zmanjšanje dimenzionalnosti z analizo glavnih komponent (PCA; ang. *Principal Component Analysis*), ipd.

A.1 Matematična formulacija

Faktorizacija SVD omogoča razgradnjo poljubne matrike v produkt treh posebnih matrik. Za poljubno matriko $A \in \mathbb{R}^{m \times n}$, $m > n$, lahko zapišemo njen singularni razcep kot:

$$A = U \Sigma V^T \quad (\text{A.1})$$

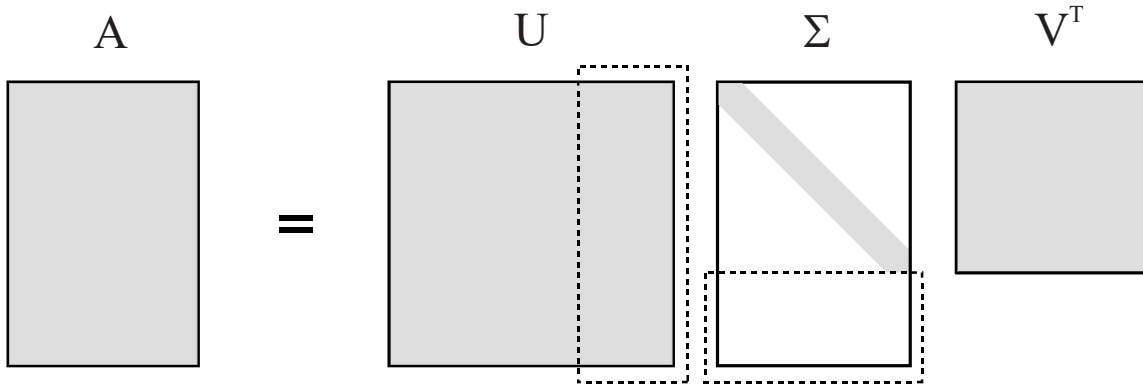
kjer so:

- $U \in \mathbb{R}^{m \times m}$ – ortogonalna matrika, katere stolpci so levi singularni vektorji matrike A ,
- $\Sigma \in \mathbb{R}^{m \times n}$ – diagonalna matrika, katere diagonalni elementi so singularne vrednosti $\sigma_i \geq 0$,
- $V \in \mathbb{R}^{n \times n}$ – ortogonalna matrika, katere stolpci so desni singularni vektorji matrike A .

Matrika Σ je praviloma zapisana v obliki:

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_r \end{bmatrix}, \quad (\text{A.2})$$

kjer so σ_i singularne vrednosti matrike A , urejene v padajočem vrstnem redu ($\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r \geq 0$). Stolpci matrik U in V so ortonormalni vektorji (ki so hkrati ortogonalni in enotski):



Slika A.1: Kanonični diagram SVD-razcepa matrike A . Stolpci matrike U so ortonormalni levi singularni vektorji, Σ je diagonalna matrika singularnih vrednosti, vrstice matrike V^T pa so ortonormalni desni singularni vektorji. Območja označena s črtkano črto predstavljajo dopolnitev matrik (ang. *padding*).

$$U = [u_1, u_2, \dots, u_m], \quad V = [v_1, v_2, \dots, v_n], \quad (\text{A.3})$$

kar pomeni, da veljata naslednji lastnosti:

$$U^T U = I_m, \quad V^T V = I_n. \quad (\text{A.4})$$

To pomeni, da stolpci U in V tvorijo ortonormalne baze za vhodni in izhodni prostor matrike A .

A.2 Povezava z lastnimi vrednostmi in lastnimi vektorji

Singularne vrednosti σ_i matrike A so povezane z njenimi lastnimi vrednostmi. Za vsako singularno vrednost σ_i obstajata enotska vektorja u_i in v_i , za katera velja:

$$Av_i = \sigma_i u_i, \quad A^T u_i = \sigma_i v_i, \quad i = 1, \dots, n. \quad (\text{A.5})$$

A.3 Lastnosti singularnega razcepa

Kanonični diagram SVD-razcepa matrike A je prikazan na sliki A.1.

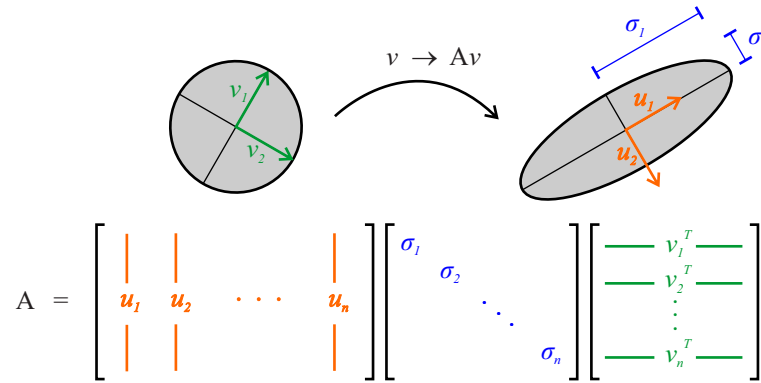
- Lastne vrednosti matrike $A^T A$ so $\sigma_1^2, \sigma_2^2, \dots, \sigma_n^2$. Lastni vektorji matrike $A^T A$ so desni singularni vektorji $v_1, v_2, \dots, v_n$.
- Lastne vrednosti matrike AA^T so $\sigma_1^2, \sigma_2^2, \dots, \sigma_n^2, 0, \dots, 0$ (slednjih $m - n$ je ničelnih, če $m > n$). Lastni vektorji matrike AA^T so levi singularni vektorji $u_1, u_2, \dots, u_m$.
- Če je $A = A^T$ (torej simetrična matrika), potem jo lahko diagonaliziramo kot:

$$A = UDU^T, \quad U^T U = I. \quad (\text{A.6})$$

V tem primeru je singularni razcep:

$$A = U\Sigma V^T, \quad \sigma_i = |\lambda_i|, \quad v_i = \text{sign}(\lambda_i)u_i, \quad (\text{A.7})$$

kjer so λ_i lastne vrednosti matrike A , pri čemer velja $\text{sign}(0) = 1$.



Slika A.2: Geometrijska interpretacija SVD za kvadratne matrice.

A.4 Geometrijska interpretacija

Če je matrika A kvadratna, torej dimenzije $m \times m$, in ima pozitivno determinanto, potem so tudi matrice U , V^T in Σ dimenzije $m \times m$. V tem primeru imajo te matrice naslednje pomeni:

- U in V^T sta ortogonalni matrici, ki predstavljata rotaciji,
- Σ je diagonalna matrika, ki predstavlja skaliranje vzdolž posameznih osi.

To pomeni, da lahko izraz $A = U\Sigma V^T$ razumemo kot zaporedje treh geometrijskih transformacij:

1. rotacija v prostoru vhodnih podatkov (množenje z V^T),
2. skaliranje vzdolž novih koordinatnih osi (množenje z Σ),
3. rotacija v končni koordinatni sistem (množenje z U).

Ta interpretacija SVD omogoča intuitivno razumevanje, kako matrika A transformira vektorje v prostoru – najprej jih rotira, nato skalira (raztegne ali skrči) vzdolž glavnih osi in na koncu ponovno rotira.

A.5 Reševanje predoločenih sistemov enačb

Predoločen sistem linearnih enačb ima obliko:

$$Ax = b, \quad A \in \mathbb{R}^{m \times n}, \quad b \in \mathbb{R}^n, \quad m > n. \quad (\text{A.8})$$

Ker sistem nima nujno eksaktne rešitve, iščemo rešitev po metodi najmanjših kvadratov (ang. *least squares*), ki minimizira normo ostanka $\|Ax - b\|_2$. Če rešitev za optimalne parametre x zapišemo z Moore-Penrose psevdoinverzom A^+ [3] in upoštevamo singularni razcep $A^+ = V\Sigma^+U^T$ dobimo:

$$x = (A^T A)^{-1} A^T b = V\Sigma^+ U^T b, \quad (\text{A.9})$$

kjer je Σ^+ *psevdoinverzna matrika* matrice Σ , ki ga je možno enostavno izračunati tako, da vzamemo elemente na diagonali matrice Σ in jih zamenjamo z njihovimi inverznimi vrednostmi:

$$\Sigma^+ = \begin{bmatrix} 1/\sigma_1 & 0 & \cdots & 0 \\ 0 & 1/\sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1/\sigma_r \end{bmatrix}, \quad (\text{A.10})$$

pri čemer je $1/\sigma_i$ definirano le za neničelne singularne vrednosti. Faktorizacija SVD omogoča stabilne rešitve tudi v primerih, ko je matrika $A^T A$ slabo pogojena ali singularna.

B. Aproximacijska 3D toga preslikava

B.1 Toga preslikava v 3D

Toga preslikava v 3D ima 6 prostih parametrov, tri translacije $\mathbf{t} = [t_x, t_y, t_z]^T$ in vzdolž 3D slike x, y, z in rotacije α (okoli osi x), β (okoli osi y) in γ (okoli osi z). Preslikovanje koordinat točke $\mathbf{p}_i = [x_i, y_i, z_i]^T$ lahko zapišemo z matričnimi operacijami tako, da rotacijski del izrazimo z Eulerjevo 3×3 rotacijsko matriko:

$$\mathbf{R} = \mathbf{R}_z(\gamma)\mathbf{R}_y(\beta)\mathbf{R}_x(\alpha) \quad (\text{B.1})$$

kjer so posamezne 3×3 rotacijske matrike:

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha \\ 0 & \sin \alpha & \cos \alpha \end{bmatrix} \quad (\text{B.2})$$

$$\mathbf{R}_y(\beta) = \begin{bmatrix} \cos \beta & 0 & \sin \beta \\ 0 & 1 & 0 \\ -\sin \beta & 0 & \cos \beta \end{bmatrix} \quad (\text{B.3})$$

$$\mathbf{R}_z(\gamma) = \begin{bmatrix} \cos \gamma & -\sin \gamma & 0 \\ \sin \gamma & \cos \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (\text{B.4})$$

Celotna kompozitna 3×3 rotacijska matrika je:

$$\mathbf{R} = \begin{bmatrix} \cos \beta \cos \gamma & \sin \alpha \sin \beta \cos \gamma - \cos \alpha \sin \gamma & \cos \alpha \sin \beta \cos \gamma + \sin \alpha \sin \gamma \\ \cos \beta \sin \gamma & \sin \alpha \sin \beta \sin \gamma + \cos \alpha \cos \gamma & \cos \alpha \sin \beta \sin \gamma - \sin \alpha \cos \gamma \\ -\sin \beta & \sin \alpha \cos \beta & \cos \alpha \cos \beta \end{bmatrix} \quad (\text{B.5})$$

Rotacije se izvajajo v zaporedju z-y-x (Tait-Bryanovi koti), zato se matrik množijo v obratnem vrstnem redu kot rotacije. Koti α , β , in γ morajo biti izraženi v radianih. Rotacijska matrika

$\mathbf{R}$ ohranja ortonormiranost ($\mathbf{R}^T \mathbf{R} = \mathbf{I}$). Ta predstavitev rotacije v 3D je še posebej uporabna pri obdelavi 3D slik in računalniškem vidu, kjer je potrebno opisati orientacijo objekta v prostoru.

B.1.1 Homogena 3D toga preslikava

Homogena 3D toga preslikava, ki vključuje rotacijo $\mathbf{R}$ (3×3 matrika) in translacijo $\mathbf{t}$ (3×1 vektor), je podana s 4×4 matriko:

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (\text{B.6})$$

kjer je $\mathbf{0} = [0, 0, 0]^T$ ničelni vektor. Preslikava 3D točke $\mathbf{p}_i = [x_i, y_i, z_i, 1]^T$ v homogenih koordinatah:

$$\mathbf{p}' = \mathbf{T} \mathbf{p}_i = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11}x_i + r_{12}y_i + r_{13}z_i + t_x \\ r_{21}x_i + r_{22}y_i + r_{23}z_i + t_y \\ r_{31}x_i + r_{32}y_i + r_{33}z_i + t_z \\ 1 \end{bmatrix} \quad (\text{B.7})$$

Inverzna transformacija:

$$\mathbf{T}^{-1} = \begin{bmatrix} \mathbf{R}^T & -\mathbf{R}^T \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \quad (\text{B.8})$$

Ključne značilnosti homogene 3D toge preslikave so, da je kombinacija rotacije in translacije predstavljena v eni 4×4 matriki, pri čemer uporaba homogenih koordinat omogoča zapis v matrični obliki. Toga 3D preslikava ohranja razdalje med točkami (togost) in kote med vektorji. Inverz 3D toge preslikave se enostavno izračuna zaradi lastnosti ortonormiranih matrik in je enak inverzu matrike dane homogene 3D toge preslikave.

B.2 Izpeljava aproksimacijske preslikave

Problem določanja aproksimacijske 3D toge preslikave v literaturi poznamo tudi pod izrazom *problem absolutne orientacije*. Problem absolutne orientacije sega v zgodnje obdobje računalniškega vida in fotogrametrije, ko je bilo potrebno najti optimalno preslikavo med dvema množicama 3D točk. Klasično rešitev sta leta 1987 neodvisno razvila Horn in Arun ter skupina soavtorjev [13], ki so predlagali zaprto obliko rešitve z uporabo razcepa po singularnih vrednostih (SVD; poglavje A). Zasnova rešitve temelji na elegantni matematični lastnosti, da se optimalna rotacijska matrika dobi z maksimizacijo sledi (ang. *trace*) kovariančne matrike med koordinatami dveh množic 3D točk. Ta pristop omogoča učinkovito izračunavanje transformacijskih parametrov brez iterativnih postopkov, kar je ključnega pomena za aplikacije, ki naj tečejo v realnem času; na primer, poravnava točkovnih oblakov, registracija medicinskih slik in robotsko vodenje. Metoda ohranja svojo aktualnost še danes zaradi svoje numerične stabilnosti in računske učinkovitosti.

Problem absolutne orientacije v kontekstu poravnave 3D kontrolnih točk rešujemo z iskanjem optimalne rotacijske matrike $\mathbf{R}$, translacijskega vektorja $\mathbf{t}$ in skalirnega faktorja s , ki preslikajo množico točk $\{\mathbf{p}_i\}$ v množico točk $\{\mathbf{q}_i\}$. Naj imamo dve množici 3D točk $\{\mathbf{p}_i\}$ in $\{\mathbf{q}_i\}$ za $i = 1, \dots, n$. Iskana preslikava naj bo:

$$\mathbf{q}_i = s\mathbf{R}\mathbf{p}_i + \mathbf{t} + \mathbf{e}_i \quad (\text{B.9})$$

kjer je $\mathbf{e}_i$ residualna napaka pri poravnavi. Cilj je minimizirati vsoto kvadratov napak:

$$\sum_{i=1}^n \|\mathbf{e}_i\|^2 = \sum_{i=1}^n \|\mathbf{q}_i - (s\mathbf{R}\mathbf{p}_i + \mathbf{t})\|^2 \rightarrow \min \quad (\text{B.10})$$

Najprej eliminiramo translacijo $\mathbf{t}$ s centriranjem točk:

$$\bar{\mathbf{p}} = \frac{1}{n} \sum_{i=1}^n \mathbf{p}_i, \quad \bar{\mathbf{q}} = \frac{1}{n} \sum_{i=1}^n \mathbf{q}_i \quad (\text{B.11})$$

Definiramo centrirane točke kot:

$$\mathbf{p}'_i = \mathbf{p}_i - \bar{\mathbf{p}}, \quad \mathbf{q}'_i = \mathbf{q}_i - \bar{\mathbf{q}} \quad (\text{B.12})$$

Nadalje rešujemo neodvisno od translacije problem rotacije in skaliranja: Minimizacijski problem postane:

$$\sum_{i=1}^n \|\mathbf{q}'_i - s\mathbf{R}\mathbf{p}'_i\|^2 \rightarrow \min \quad (\text{B.13})$$

Razvijemo kvadratno obliko:

$$\sum \|\mathbf{q}'_i\|^2 - 2s \sum \mathbf{q}'_i{}^T \mathbf{R}\mathbf{p}'_i + s^2 \sum \|\mathbf{p}'_i\|^2 \rightarrow \min \quad (\text{B.14})$$

Optimalno skaliranje določimo z odvajanjem enačbe (B.14) po s in iskanjem ničelne točke:

$$s = \frac{\sum \mathbf{q}'_i{}^T \mathbf{R}\mathbf{p}'_i}{\sum \|\mathbf{p}'_i\|^2} \quad (\text{B.15})$$

Dobljen rezultat v enačbi (B.15) vstavimo s nazaj in dobimo izraz za minimizacijo, kjer je edina spremenljivka $\mathbf{R}$:

$$\sum \|\mathbf{q}'_i\|^2 - \frac{(\sum \mathbf{q}'_i{}^T \mathbf{R}\mathbf{p}'_i)^2}{\sum \|\mathbf{p}'_i\|^2} \rightarrow \min \quad (\text{B.16})$$

Problem minimizacije v enačbi (B.16) je ekvivalenten maksimizaciji:

$$\sum \mathbf{q}'_i{}^T \mathbf{R}\mathbf{p}'_i = \text{tr}(\mathbf{R}^T \mathbf{H}) \rightarrow \max \quad (\text{B.17})$$

kjer je $\mathbf{H} = \sum \mathbf{q}'_i \mathbf{p}'_i{}^T$. Rešitev dobimo z uporabo SVD (poglavje A). Razcep $\mathbf{H} = \mathbf{U}\Sigma\mathbf{V}^T$ po singularnih vrednostih uporabimo, da zapišemo optimalno rotacijo:

$$\mathbf{R} = \mathbf{V}\mathbf{U}^T \quad (\text{B.18})$$

Če $\det(\mathbf{R}) = -1$, popravimo rešitev z zrcaljenjem:

$$\mathbf{V}' = \mathbf{V} \cdot \text{diag}(1, 1, \dots, 1, -1), \quad \mathbf{R} = \mathbf{V}'\mathbf{U}^T \quad (\text{B.19})$$

Končna rešitev, s skaliranjem in translacijo je podana kot:

$$s = \frac{\text{tr}(\Sigma)}{\sum \|\mathbf{p}'_i\|^2}, \quad \mathbf{t} = \bar{\mathbf{q}} - s\mathbf{R}\bar{\mathbf{p}} \quad (\text{B.20})$$

B.3 Aproximacijska preslikava v računskih korakih

Za implementacijo aproksimacijske preslikave uporabimo naslednjih 7 (podobnostna preslikava) oziroma 6 (toga preslikava, brez koraka 6.) računskih korakov:

1. Izračun težišč:

$$\bar{\mathbf{p}} = \frac{1}{n} \sum_{i=1}^n \mathbf{p}_i, \quad \bar{\mathbf{q}} = \frac{1}{n} \sum_{i=1}^n \mathbf{q}_i \quad (\text{B.21})$$

2. Centriranje točk:

$$\mathbf{p}'_i = \mathbf{p}_i - \bar{\mathbf{p}}, \quad \mathbf{q}'_i = \mathbf{q}_i - \bar{\mathbf{q}} \quad (\text{B.22})$$

3. Izračun kovariančne matrike:

$$\mathbf{H} = \sum_{i=1}^n \mathbf{q}'_i \mathbf{p}'_i{}^T \quad (\text{B.23})$$

4. Razcep po singularnih vrednostih (SVD):

$$\mathbf{H} = \mathbf{U}\Sigma\mathbf{V}^T \quad (\text{B.24})$$

5. Določitev rotacijske matrike:

$$\mathbf{R} = \mathbf{V}\mathbf{U}^T \quad (\text{B.25})$$

Če $\det(\mathbf{R}) = -1$, potem matriko popravimo:

$$\mathbf{V}' = \mathbf{V} \cdot \text{diag}(1, 1, \dots, 1, -1), \quad \mathbf{R} = \mathbf{V}'\mathbf{U}^T \quad (\text{B.26})$$

6. Izračun skalirnega faktorja (*izpustimo, če iščemo le togo preslikavo*):

$$s = \frac{\sum_{i=1}^n \mathbf{q}'_i{}^T \mathbf{R} \mathbf{p}'_i}{\sum_{i=1}^n \|\mathbf{p}'_i\|^2} \quad (\text{B.27})$$

7. Določitev translacije (*uporabimo $s=1$, če iščemo le togo preslikavo*):

$$\mathbf{t} = \bar{\mathbf{q}} - s\mathbf{R}\bar{\mathbf{p}} \quad (\text{B.28})$$

V koraku 5. dobimo rotacijsko matriko $\mathbf{R}$, v koraku 7. pa translacijski vektor $\mathbf{t}$. **Toga preslikava** točk je nato podana z:

$$\mathbf{q}_i = \mathbf{R}\mathbf{p}_i + \mathbf{t} \quad (\text{B.29})$$

Če v gornjih korakih določimo tudi parameter skaliranja s , potem lahko zapišemo **podobnostno preslikavo** (toga preslikava + skaliranje) točk kot:

$$\mathbf{q}_i = s\mathbf{R}\mathbf{p}_i + \mathbf{t} \quad (\text{B.30})$$



C. Razvojna okolja

Python je eden najbolj priljubljenih programskih jezikov na svetu, ki se zaradi svoje enostavnosti, berljivosti in širokega nabora uporabnih in učinkovitih knjižnic pogosto uporablja v razvoju programske opreme, podatkovne analize in posebej na področju računalniškega vida. Po podatkih TIOBE Indexa¹, ki meri priljubljenost programskih jezikov, je Python v zadnjih letih redno med prvimi tremi, pogosto celo na prvem mestu, kar kaže na njegovo široko sprejetost v industriji in akademskih krogih [28]. Na področju računalniškega vida je Python še posebej privlačen zaradi bogatega ekosistema knjižnic, kot so OpenCV, PyTorch in TensorFlow, ki olajšajo obdelavo slik, globoko učenje in analizo vizualnih podatkov.

Razvojni proces v Pythonu zahteva učinkovita orodja, ki omogočajo hitro namestitve, konfiguracijo in upravljanje knjižnic, saj je uspeh projekta odvisen od stabilnega in reproducibilnega razvojnega okolja. Brez pravilno nastavljenih orodij lahko pride do težav, kot so nezdržljive verzije knjižnic, težave s so-odvisnostmi knjižnic ali pomanjkanje zmogljivosti za zahtevne naloge, kot je učenje in poganjanje globokih modelov za računalniški vid.

V tem poglavju bomo pregledali ključna razvojna orodja, ki so uporabna na vseh operacijskih sistemih (Windows, MacOS in Linux) in so posebej koristna za razvoj na področju računalniškega vida. Obravnavali bomo naslednja orodja:

- **Google Colab:** Spletno okolje za Python, ki ponuja brezplačen dostop do grafičnih in tenzorskih procesnih enot (GPU; ang. *Graphic Processing Unit*; TPU; ang. *Tensor Processing Unit*), je idealno za hitre teste in medsebojno deljenje projektov.
- **Visual Studio Code (VS Code):** Močno lokalno razvojno okolje s podporo za Conda in Docker kontejnerje, ki omogoča prilagodljivo delo na projektih.
- **Dodatna orodja:** Jupyter Notebook za interaktivni razvoj, virtualna okolja (venv) za izolacijo projektov, ter ključne knjižnice za računalniški vid, kot je OpenCV.

Za dodatne vire in literaturo glede jezika Python priporočamo “Uvod v programski jezik Python” avtorja Mirana Bürmena [2], ki podaja pregled osnovne sintakse jezika, pregled temeljnih knjižnic kot sta numpy in matplotlib in primere njune uporabe. Za področje računalniškega (in robotskega) vida priporočam knjigo “Programming Computer Vision with Python” avtorja

¹(TIOBE Index: <https://www.tiobe.com/tiobe-index/>)

Jan Erik Solem [7], ki ponuja poglobljen uvod v računalniški vid s praktičnimi primeri v jeziku Python. Uporabni spletni viri so uradna dokumentacija Pythona (<https://docs.python.org/>) in vadbice na spletnih straneh, kot so Real Python (<https://realpython.com/>) in PyImageSearch (<https://www.pyimagesearch.com/>).

C.1 Osnovna namestitvev in konfiguracija Pythona

Za začetek dela s Pythonom je potrebna pravilna namestitev in konfiguracija, ki zagotavlja stabilno in funkcionalno razvojno okolje. V tem podpoglavju bomo podali navodila za namestitev Pythona, uporabo upravljalnika paketov pip in nastavitev poti (PATH) na različnih operacijskih sistemih, ter pokazali, kako preveriti namestitev.

C.1.1 Namestitev Pythona

Python lahko namestite na dva načina: preko uradne strani <https://www.python.org/downloads/> ali z uporabo Anaconda distribucije, ki je posebej priporočljiva za razvoj na področju računalniškega vida, saj vključuje mnoge priljubljene knjižnice, kot so NumPy, Matplotlib in SciPy, ter upravljalnik paketov Conda.

- **Namestitev preko python.org:** Obiščite <https://www.python.org/downloads/> in prenesite izbrano verzijo Pythona (npr. 3.9 ali novejšo). Med namestitvijo na Windows in macOS izberite možnost “Add Python to PATH”, kar olajša dostop do Pythona iz terminala oziroma ukazne vrstice. Na Linuxu običajno uporabite upravljalnik paketov sistema, npr.:

```
$ sudo apt-get update
$ sudo apt-get install python3 python3-pip # Na Debian/Ubuntu
```

- **Namestitev s Conda:** Namestitev Conda na posamezne operacijske sisteme je opisna na [https://docs.conda.io](https://docs.conda.io/en/latest/miniconda.html) → Install. Prenesite Miniconda z <https://docs.conda.io/en/latest/miniconda.html> ali Anaconda z <https://www.anaconda.com/products/distribution>. Namestite ga z privzetimi nastavitvami. Na primer, v Linux okolju izvedete namestitev z naslednjimi ukazi v ukaznem oknu:

```
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh
```

Anaconda vključuje Python, Conda in večino potrebnih knjižnic za računalniški vid, kar prihrani čas pri ročni namestitvi, medtem, ko je potrebno pri Miniconda ročno namestiti knjižnice.

Za prikaz delovanja rešitev vaj v tem priročniku je bila uporabljena različica “Python 3.12.7”, na MacOS sistemu, z namestitvijo preko [python.org](https://www.python.org).

C.1.2 Preverjanje namestitve

Poleg ukaza python potrebujete upravljalnik paketov pip, ki je standardno orodje za namestitev Python knjižnic. Po namestitvi Pythona ali Anaconde preverite, ali je pip na voljo, in nastavite PATH, da lahko izvajate ukaze iz katere koli mape.

- **Na Windows:** Po namestitvi Pythona preko [python.org](https://www.python.org) preverite, ali je PATH nastavljen. Odprite Command Prompt in vnesite:

```
C:\> python --version
Python 3.9.7

C:\> pip --version
pip 21.2.3 from C:\Python39\Lib\site-packages\pip (python 3.9)
```

Če ukazi ne delujejo, morate ročno dodati pot do Pythona (npr. C:\Python39\;C:\Python39\Scripts\)) v sistemsko spremenljivko PATH preko nastavitev sistema.

- **Na macOS in Linux:** Terminal običajno že vključuje Python in pip, ker sta nameščena kot del operacijskega sistema. Preverite z:

```
$ python3 --version
Python 3.9.7

$ pip3 --version
pip 21.2.3 from /usr/local/lib/python3.9/site-packages/pip (python 3.9)
```

Če uporabljate Anaconda, uporabite Conda namesto pip za večjo stabilnost:

```
$ conda --version
conda 23.7.4
```

Nastavitve PATH na macOS/Linux ni potrebna, če ste namestili Python v standardno lokacijo ali če uporabljate Anaconda.

C.2 Upravljanje Python razvojnih okolij s pip

V tem podpoglavju bomo podrobno opisali, kako uporabljati pip, namen, vsebino in pripravo datoteke `requirements.txt`, izbiro specifičnih verzij knjižnic, razreševanje konfliktov med odvisnostmi in druge pomembne vidike. pip omogoča namestitev Python paketov iz PyPI (ang. *Python Package Index*). Osnovni ukaz za namestitev paketa je `pip install <ime_paketa>`. Primer namestitve knjižnice `numpy`:

```
$ pip install numpy
Collecting numpy
  Downloading numpy-1.21.0-cp39-cp39-win_amd64.whl (13.0 MB)
    |.....| 13.0 MB 2.5 MB/s
Installing collected packages: numpy
Successfully installed numpy-1.21.0
```

Za posodabljanje paketa uporabite:

```
$ pip install --upgrade numpy
Collecting numpy
  Downloading numpy-1.22.0-cp39-cp39-win_amd64.whl (14.1 MB)
    |.....| 14.1 MB 3.0 MB/s
Installing collected packages: numpy
  Attempting uninstall: numpy
    Found existing installation: numpy 1.21.0
    Uninstalling numpy-1.21.0:
      Successfully uninstalled numpy-1.21.0
  Successfully installed numpy-1.22.0
```

Za odstranitev paketa uporabite:

```
$ pip uninstall numpy
Found existing installation: numpy 1.22.0
Uninstalling numpy-1.22.0:
  Would remove:
    c:\python39\lib\site-packages\numpy-1.22.0.dist-info\*
    c:\python39\lib\site-packages\numpy\*
Proceed (y/n)? y
Successfully uninstalled numpy-1.22.0
```

C.2.1 Namen in uporaba requirements.txt

Datoteka requirements.txt je ključna za dokumentiranje in reproduciranje odvisnosti projekta. Vsebuje seznam programskih knjižnic z njihovimi verzijami, ki jih projekt zahteva. Primer vsebine requirements.txt:

```
numpy==1.21.0
pandas>=1.3.5
opencv-python<4.6
```

Za namestitev vseh paketov, navedenih v requirements.txt, uporabite:

```
$ pip install -r requirements.txt
Collecting numpy==1.21.0
  Downloading numpy-1.21.0-cp39-cp39-win_amd64.whl (13.0 MB)
    |.....| 13.0 MB 2.5 MB/s
Collecting pandas>=1.3.5
  Downloading pandas-1.3.5-cp39-cp39-win_amd64.whl (10.2 MB)
    |.....| 10.2 MB 3.0 MB/s
Collecting opencv-python<4.6
  Downloading opencv-python-4.5.5.64-cp39-cp39-win_amd64.whl (35.4 MB)
    |.....| 35.4 MB 2.1 MB/s
Installing collected packages: numpy, pandas, opencv-python
Successfully installed numpy-1.21.0 pandas-1.3.5 opencv-python-4.5.5.64
```

Za ustvarjanje requirements.txt iz trenutno nameščenih paketov uporabite:

```
$ pip freeze > requirements.txt
```

Ta ukaz ustvari datoteko z vsemi nameščenimi paketi in njihovimi verzijami.

Za pripravo ustreznega razvojnega Python okolja za izvedbo vaj v tem priročniku je v javno dostopnem Git repozitoriju dodana datoteka requirements.txt, ki vključuje skupno 52 Python knjižnic s specifičnimi verzijami. Večina od teh 52 knjižnic predstavlja so-odvisnosti od minimalnega nabora knjižnic, ki jih potrebujete za izvedbo vaj. Minimalni nabor knjižnic lahko naložite z naslednjim ukazom:

```
pip install numpy scipy scikit-image scikit-learn pandas pillow opencv-python
↪ opencv-contrib-python matplotlib
```

Orodje pip (ali conda) bo nato naložilo še vse ostale so-odvisne knjižnice.

C.2.2 Izbor verzij knjižnic

Pri izbiri specifičnih verzij je pomembno zagotoviti skladnost in stabilnost. Lahko specificirate:

- Točno verzijo: ==1.21.0 (npr. numpy==1.21.0).
- Minimalno verzijo: >=1.21.0 (npr. pandas>=1.3.5).
- Maksimalno verzijo: <4.6 (npr. opencv-python<4.6).

Če želite preveriti, katere verzije so na voljo, uporabite:

```
$ pip search numpy
numpy (1.22.0) - NumPy: multidimensional array processing for numbers, strings,
    ↪ records, and objects.
INSTALLED: 1.21.0
LATEST:    1.22.0
```

Konflikti se pojavijo, ko različni paketi zahtevajo nezdružljive verzije istih knjižnic. Na primer, če en paket zahteva numpy==1.20.0, drugi pa numpy==1.22.0, pip lahko vrne napako:

```
$ pip install package1 package2
ERROR: Cannot install package1 and package2 because these package versions have
➔ conflicting dependencies.

The conflict is caused by:
  package1 1.0 depends on numpy==1.20.0
  package2 2.0 depends on numpy==1.22.0
```

Za razreševanje ročno uredite `requirements.txt` in izberite kompatibilno verzijo (npr. najnovejšo stabilno verzijo, ki jo podpirata oba paketa). Nato uporabite `pip install -no-cache-dir` za ponovno preverjanje ali `pipdeptree` za analizo odvisnosti:

```
$ pip install pipdeptree
$ pipdeptree
numpy==1.21.0
- package1 [requires numpy==1.20.0]
- package2 [requires numpy==1.22.0]
```

Prav zaradi morebitnih različnih zahtev glede verzij knjižnic je smiselno za posamezen Python projekt ustvariti lastno razvojno okolje in namestiti ustrezne pakete. Za ta namen predlagamo uporabo virtualnega okolja (venv), ki je ločeno od sistemske Python namestitve (nujno na MacOS in Linux). Ključni ukazi za ustvarjanje in aktivacijo virtualnega okolja so:

```
$ python -m venv myenv
$ source myenv/bin/activate # Na Linux/macOS
$ myenv\Scripts\activate    # Na Windows
(myenv) $ pip install opencv-python numpy
```

Alternativno lahko uporabite Conda za ločeno upravljanje Python namestitve in odvisnosti.

V življenjskem ciklu razvoja in posodabljanja programske kode Python redno posodablajte verzije knjižnic `pip install -upgrade pip`. Če se srečujete s težavami, preverite dokumentacijo PyPI (<https://pypi.org>) ali forum, kot je Stack Overflow (<https://stackoverflow.com/>) ali povprašajte za nasvet klepetalnega bota (<https://chatgpt.com> ali <https://chat.deepseek.com/>). S temi koraki lahko učinkovito upravljate odvisnosti v Python projektih, kar je ključno za stabilnost in reproducibilnost vašega dela.

C.3 Upravljanje Python razvojnih okolij s conda

Ukaz `conda` uporabljamo podobno kot ukaz `pip`, vendar omogoča več funkcij v enem: (i) ustvarjanje in manipuliranje virtualnih okolij, (ii) upravljanje z verzijami Python knjižnic, (iii) samodejno ali ročno razreševanje konfliktov med so-odvisnostmi knjižnic, ipd. V terminalu oziroma ukazni vrstici ustvarite novo virtualno okolje in naložite Python knjižnice kot:

```
$ conda create -n myenv python=3.9
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: C:\Users\YourUser\miniconda3\envs\myenv

Proceed ([y]/n)? y

Preparing transaction: done
Verifying transaction: done
```

```

Executing transaction: done
#
# To activate this environment, use:
#
#     $ conda activate myenv
#
# To deactivate an active environment, use:
#
#     $ conda deactivate
#

```

Nato aktivirajte okolje z:

```

$ conda activate myenv
(myenv) $

```

Namestite ključne knjižnice za robotski vid:

```

(myenv) $ conda install opencv numpy pytorch torchvision -c pytorch
Collecting package metadata (current_repodata.json): done
Solving environment: done
## Package Plan ##

The following packages will be downloaded:


```

package	build		
opencv-4.5.5	py39h4f2741e_0	14.2 MB	conda-forge
numpy-1.21.0	py39h2e03b76_0	5.5 MB	conda-forge
pytorch-1.10.0	py3.9_cuda11.1_cudnn8_0	609.9 MB	pytorch
torchvision-0.11.0	py39_cu111	6.8 MB	pytorch
Total:		636.4 MB	

```

Proceed ([y]/n)? y

Downloading and Extracting Packages
...
Preparing transaction: done
Verifying transaction: done
Executing transaction: done

```

Shranite `requirements.txt` za dokumentiranje so-odvisnosti knjižnic:

```

(myenv) $ pip freeze > requirements.txt

```

Ta datoteka bo vsebovala seznam vseh nameščenih paketov, na primer:

```

numpy==1.21.0
opencv-python==4.5.5.64
pytorch==1.10.0
torchvision==0.11.0

```

C.4 Uporaba Docker virtualnega okolja

Docker kontejnerji (ang. *Docker containers*) omogočajo izolacijo in reproducibilnost razvojnega okolja, ne glede na operacijski sistem in Python različico oziroma distribucijo. Kontejnerji so

izolirana virtualna okolja, ki vsebujejo vse potrebno za poganjanje aplikacije (kodo, knjižnice). Uporabni so za zagotavljanje, da projekt deluje enako na različnih operacijskih sistemih in da je med njimi enostavno prenosljiv.

Namestitev Dockerja izvedete tako, da prenesete Docker Desktop z <https://www.docker.com/products/docker-desktop> za Windows/macOS ali Docker CE za Linux. Sledite navodilom za namestitev.

C.4.1 Razumevanje osnovnih Docker konceptov

Poleg Docker kontejnerjev, ki predstavljajo *dinamično razvojno okolje* poznamo še “Docker slike” (ang. *Docker images*), ki predstavljajo *zamrznjeno razvojno okolje*. Za trajno hrambo podatkov uporabljamo “Docker mape” (ang. *Docker volume*). Vsaka Docker slika ob *zagonu* z izbranimi parametri postane “docker kontejner”, ki lahko (ni pa nujno) za hrambo podatkov uporablja “docker mapo”. Razlaga konceptov treh omenjenih gradnikov je podana v nadaljevanju:

Docker slika

:

- **Opis:** Nespremenljiva predloga z navodili za ustvarjanje kontejnerjev
- **Analogija:** Kot ISO datoteka za virtualni operacijski sistem
- **Značilnosti:**
 - Sestavljena iz več slojev (ang. *layers*)
 - Vsebuje vse potrebne odvisnosti in nastavitve
 - Ustvarja se z `Dockerfile` ali `docker commit`
- **Primer ukaza:**

```
docker build -t moja-slika .
```

Docker kontejner

:

- **Opis:** Izvedljiva instanca Docker slike
- **Analogija:** Kot zagon programa iz ISO datoteke
- **Značilnosti:**
 - Izolirano okolje z lastnim procesnim prostorom
 - Lahko se ustvari, zaustavi, odstrani ali ponovno zažene
 - Volatilen – spremembe se privzeto ne shranijo po odstranitvi
- **Primer ukaza:**

```
docker run -d --name moj-kontejner moja-slika
```

Docker mapa

:

- **Opis:** Mehanizem za trajno shranjevanje podatkov
- **Analogija:** Zunanji disk za kontejner
- **Značilnosti:**
 - Ohranja podatke neodvisno od življenjskega cikla kontejnerja
 - Lahko se uporablja za deljenje podatkov med kontejnerji
 - Obstajajo tri vrste: imenovani (named), anonimni in gostiteljevi (host)
- **Primer ukaza:**

```
docker volume create moj-volume
```


Primerjalna tabela

Koncept	Namen	Trajnost	Primer uporabe
Docker slika	Predloga za kontejner	Trajna	Osnovna Python slika
Docker kontejner	Izvedba aplikacije	Volatilna	Delujoča spletna aplikacija
Docker mapa	Shranjevanje podatkov	Trajna	Podatkovna baza

- **Povezava med koncepti:** Slika → Kontejner + Volume = Delujoča aplikacija s trajnimi podatki.
- **Pomembno:** Kontejnerji so “žive oziroma delujoče” slike, volumni pa omogočajo trajnost podatkov.

C.4.2 Ustvarjanje virtualnega okolja

Recept za pripravo virtualnega okolja se pripravi z zapisom zaporedja ukazov v ukazni vrstici, kot je navedeno v poglavjih C.1, C.2 in C.3. Te ukaze zapišemo v datoteko, ki jo običajno poimenujemo Dockerfile.

Pomen osnovnih Docker ukazov Docker uporablja posebne ukaze v Dockerfile za definiranje postopka gradnje kontejnerja. Spodaj je razlaga ključnih ukazov:

FROM

- **Namen:** Določi osnovno sliko (base image) za kontejner
- **Sintaksa:** FROM <image>[:<tag>]
- **Primer:**

```
FROM python:3.9-slim
```

- **Podrobnosti:**
 - Vedno mora biti prvi ukaz v Dockerfile
 - Oznaka (ang. *tag*) določa verzijo (npr. 3.9, latest)
 - Uradne slike najdemo na <https://hub.docker.com>

WORKDIR

- **Namen:** Nastavi delovni imenik znotraj kontejnerja
- **Sintaksa:** WORKDIR /path/to/directory
- **Primer:**

```
WORKDIR /app
```

- **Podrobnosti:**
 - Ustvari imenik, če ne obstaja
 - Vpliva na vse naslednje ukaze (COPY, RUN, itd.)
 - Alternativa ukazu cd v terminalu

COPY

- **Namen:** Kopira datoteke iz gostitelja v kontejner
- **Sintaksa:** COPY <src> <dest>
- **Primer:**

```
COPY requirements.txt .
COPY src/ ./src/
```

- **Podrobnosti:**
 - Relativne poti so glede na WORKDIR

- Podpira vzorce (npr. *.py kopira vse datoteke z navedeno končnico)
- Za večjo fleksibilnost obstaja tudi ADD

RUN

- **Namen:** Izvede ukaze med gradnjo slike
- **Sintaksa:** RUN <command>
- **Primer:**

```
RUN apt-get update && apt-get install -y git
RUN pip install -r requirements.txt
```

- **Podrobnosti:**
 - Vsak RUN ustvari nov sloj (layer) v sliki
 - Priporočljivo je združevati povezane ukaze z &&
 - Izvaja se v privzetem terminalu (/bin/sh)

CMD

- **Namen:** Določi privzeti ukaz za zagon kontejnerja
- **Sintaksa:**
 - CMD command param1 param2 (format ukazne vrstice)
 - CMD ["executable", "param1", "param2"] (strukturiran format ukazne vrstice, kjer so ukazi ali parametri navedeni ločeno z vejicami)
- **Primer:**

```
CMD ["python", "app.py"]
```

- **Podrobnosti:**
 - Lahko ga prepišemo z argumenti pri docker run
 - V Dockerfile je dovoljen samo en CMD
 - Če ni podan, se uporabi CMD iz osnovne slike

ENTRYPOINT

- **Namen:** Določi fiksni ukaz, ki se izvede ob zagonu
- **Sintaksa:**
 - ENTRYPOINT ["executable", "param1"] (exec forma)
 - ENTRYPOINT command param1 (shell forma)
- **Primer:**

```
ENTRYPOINT ["python"]
CMD ["app.py"]
```

- **Podrobnosti:**
 - Argumenti docker run se dodajo k ENTRYPOINT
 - Če ni podan CMD, morajo biti vsi argumenti podani ob zagonu
 - Uporaben za "wrappere" okoli aplikacij

Primerjalna tabela: CMD vs ENTRYPOINT

Lastnost	CMD	ENTRYPOINT
Namen	Privzeti parametri	Fiksni izvedljivi ukaz
Prepisovanje	Da (z argumenti docker run)	Ne (le dodajanje argumentov)
Priporočena uporaba	Spremenljivi parametri	Fiksna osnova ukaza

Primer kombinacije ENTRYPOINT in CMD v Dockerfile:

```
ENTRYPOINT ["python"]
CMD ["--help"] # Privzeto pokaže help, če ni argumentov v docker run
```

Na primer, ustvarite datoteko `Dockerfile` v korenu projekta in zapišite ukaze:

```
FROM python:3.9-slim

WORKDIR /app
COPY /pot/do/moj_program.py .
COPY requirements.txt .
RUN pip install -r requirements.txt

CMD ["python", "moj_program.py"]
```

Ukaz `FROM python:3.9-slim` določa osnovno Python 3.9 Docker sliko, optimizirano za majhno velikost (glej C.4.2). Nato `WORKDIR /app` nastavi delovni imenik v kontejnerju na `/app`. Z ukazoma `COPY /pot/do/moj_program.py .` in `COPY requirements.txt .` kopiramo programske kodo in seznam odvisnosti v kontejner. Ukaz `RUN pip install -r requirements.txt` namesti vse potrebne Python pakete iz datoteke `requirements.txt`. Nazadnje ukaz `CMD ["python", "moj_program.py"]` določi privzeti ukaz za zagon programa ob inicializaciji kontejnerja. Ta datoteka *Dockerfile* naj z ustreznim zaporedjem ukazov, kot bi jih izvedli v terminali oziroma ukazni vrstici, ustvari optimizirano okolje za izvajanje Python aplikacije z vsemi potrebnimi Python knjižnicami in vašimi programi.

Nato lahko zgradite Docker sliko z:

```
$ docker build -t myimage .
Sending build context to Docker daemon 2.048kB
Step 1/4 : FROM python:3.9-slim
--> 48e5f4513959
Step 2/4 : WORKDIR /app
--> Using cache
--> 6d7f8b3a9a0c
Step 3/4 : COPY requirements.txt .
--> Using cache
--> 9a3b2c1d8e7f
Step 4/4 : RUN pip install -r requirements.txt
--> Running in 4b5c6d7e8f9a
Collecting numpy==1.21.0 (from -r requirements.txt (line 1))
  Downloading numpy-1.21.0-cp39-cp39-manylinux_2_17_x86_64.manylinux2014_x86_64.
    ↪ whl (15.8 MB)
    |.....| 15.8 MB 3.2 MB/s
...
Successfully built myimage
```

in izvedete zagon kontejnerja:

```
$ docker run -it myimage
Python 3.9.7 (default, Sep 16 2021, 08:50:19)
[GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Glavne nastavitve za zagon kontejnerjev z ukazom `docker run` vključujejo: `-i` (interaktivni način, omogoča vnos preko STDIN), `-t` (alocira psevd-terminal za interaktivne seje, običajno uporabljen skupaj z `-i` kot `-it`), `-d` (detached način, poganja kontejner v ozadju), `-rm` (samodejno

odstrani kontejner po izhodu), -p (preslikanje vrat, npr. -p 8080:80), -v (montiranje volumnov, npr. -v /host:/container), in -name (določitev imena kontejnerja). Te opcije omogočajo fleksibilno upravljanje vedenja kontejnerjev med zagonom. Ključna prednost, ko enkrat preverimo delovanje para Docker slike in kontejnerja, je reproducibilnost (ista slika oz. kontejner deluje povsod), enostavno deljenje okolja z drugimi razvijalci in izolacija od lokalnega sistema.

Uporaba vnaprej pripravljenih Docker slik

Uporabite ali prilagodite lahko različne osnovne Docker slike, ki so na voljo na <https://hub.docker.com/>. Za pridobitev Docker slike iz registra Docker Hub uporabite ukaz `docker pull`. Osnovna sintaksa je:

```
docker pull <ime_slike>:<oznaka>
```

Primer za pridobitev uradne Python slike verzije 3.9:

```
docker pull python:3.9-slim
```

Če oznaka ni podana, se privzeto uporabi `latest`. Slika se prenese lokalno in jo lahko preverite z:

```
docker images
```

C.5 Uporaba Docker mape

Za uporabo Docker mape ali sistemske mape pri zagonu kontejnerja uporabimo zastavico `-v` ali `-mount`. Glavne razlike:

- `-v` je krajša sintaksa za osnovno uporabo
- `-mount` ponuja več možnosti za naprednejše scenarije

Osnovna sintaksa je naslednja:

```
docker run -v ime_volumna:/pot/v/kontejnerju slika
```

Primer uporabe obstoječe docker mape:

```
# Ustvari mapo (če še ne obstaja)
docker volume create moj_volume

# Poženi kontejner z montirano mapo
docker run -d -v moj_volume:/app/podatki nginx
```

Primer uporabe sistemske mape:

```
docker run -v /absolutna/pot/na/pc:/pot/v/kontejnerju slika
```

Pri uporabi sistemskih map morajo biti nastavljene ustrezne pravice za branje/pisanje/izvajanje datotek in map tako na sistemu kot v Docker kontejnerju.

Uporaba mape z omejitvami, na primer samo za branje (ang. *read-only*):

```
docker run -v moj_volume:/podatki:ro alpine
```

Preverjanje seznama Docker map v uporabi:

```
# Pokaže podrobnosti o kontejnerju (vključno z volumni)
docker inspect ime_kontejnerja

# Pokaže vse volume
docker volume ls
```

C.5.1 Nastavitev Docker kontejnerja za daljinski dostop

Daljinski dostop z “varno lupino” (SSH; ang. *Secure Shell*) je varen protokol za upravljanje in dostop do oddaljenih računalnikov ali strežnikov preko omrežja. Uporablja šifrirano povezavo za izmenjavo podatkov, kar zagotavlja varnost komunikacije, vključno z gesli in ukazi. S pomočjo ssh klienta lahko razvijalci dostopajo do terminala oddaljenega sistema, kot je Docker kontejner, izvajajo ukaze, upravljajo datoteke in izvajajo programe, kot so Python skripte za računalniški vid, ne da bi fizično sedeli pred tem sistemom. SSH je široko uporabljen zaradi svoje robustnosti in enostavnosti uporabe, zlasti v razvojnih in produkcijskih okoljih.

SSH

Več informacij o SSH, njegovi namestitvi in uporabi, najdete na uradni strani OpenSSH projekta, dostopna na naslovu <https://www.openssh.com/>. Ta stran ponuja uradno dokumentacijo, novice in vire o OpenSSH, ki je najbolj razširjena implementacija SSH protokola. Na voljo so vodniki za namestitev na različnih operacijskih sistemih (Windows, macOS, Linux), konfiguracijo strežnika in klienta, ter napredne teme, kot so uporaba SSH ključev, tuneliranje in varnostne nastavitve. Poleg tega vključuje povezave do prenosov, FAQ in referenčno dokumentacijo (man pages). Za dodatne praktične primere priporočamo tudi vadnice na DigitalOcean ter dokumentacijo na <https://docs.github.com/en/authentication/connecting-to-github-with-ssh> spletni strani GitHub, zlasti če uporabljate SSH v kontekstu Git ali Dockerja. Ti viri so zanesljivi, široko uporabljeni in redno posodobljeni.

Za omogočanje daljinskega dostopa do Docker kontejnerja preko SSH, morate prilagoditi Dockerfile in konfigurirati SSH strežnik. Poleg tega lahko kontejner zaženete v odcepljenem (ang. *detached*) načinu, kar omogoča, da teče v ozadju, medtem ko ohranjate možnost dostopa preko SSH.

Konfiguracija Dockerfile za SSH

Ustvarite Dockerfile z naslednjo vsebino, ki temelji na Python sliki in doda SSH podporo:

```
FROM python:3.9-slim

# Namestitev OpenSSH strežnika in drugih orodij
RUN apt-get update && apt-get install -y openssh-server
RUN mkdir /var/run/sshd

# Ustvarjanje uporabnika in nastavitve gesla
RUN useradd -m -s /bin/bash developer && \
    echo "developer:password123" | chpasswd

# Konfiguracija SSH
RUN echo "PermitRootLogin no" >> /etc/ssh/sshd_config && \
    echo "PasswordAuthentication yes" >> /etc/ssh/sshd_config && \
    echo "AllowUsers developer" >> /etc/ssh/sshd_config

# Omogočanje dostopa do SSH porta
EXPOSE 22

# Kopiranje requirements.txt in namestitev Python paketov
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt

# Zagon SSH strežnika pri zagonu kontejnerja
```

```
CMD ["/usr/sbin/sshd", "-D"]
```

Nato zgradite sliko z:

```
$ docker build -t python_ssh_image .
Sending build context to Docker daemon 2.048kB
Step 1/9 : FROM python:3.9-slim
--> 48e5f4513959
Step 2/9 : RUN apt-get update && apt-get install -y openssh-server
--> Running in 9a3b2c1d8e7f
...
Successfully built python_ssh_image
```

Za zagon kontejnerja v “odcepljenem” načinu (ang. *detached mode*, omogočimo s stikalom `docker run -d ...`) uporabite naslednji ukaz, ki omogoča, da kontejner teče v ozadju in izpostavi port 22 za SSH:

```
$ docker run -d -p 2222:22 python_ssh_image
a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0u1v2w3x4y5z6
```

Tu ‘-d’ zahteva zagon kontejnerja v odcepljenem načinu, medtem ko ‘-p 2222:22’ preusmeri lokalni port 2222 na port 22 v kontejnerju (lahko izberete drug port, npr. 2222, a različen od 22 da se izognemo konfliktom z lokalnim SSH). Za povezavo preko SSH iz drugega terminala ali računalnika uporabite:

```
$ ssh developer@localhost -p 2222
The authenticity of host '[localhost]:2222 (:::1):2222)' can't be established.
ECDSA key fingerprint is SHA256:...
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[localhost]:2222' (ECDSA) to the list of known hosts
↪ .
developer@localhost's password:
password123
Welcome to Python SSH Container!
$
```

Ko ste povezani, lahko v kontejnerju izvajate ukaze, kot so zagon Python skript, namestitve dodatnih paketov ali obdelava podatkov za računalniški vid.

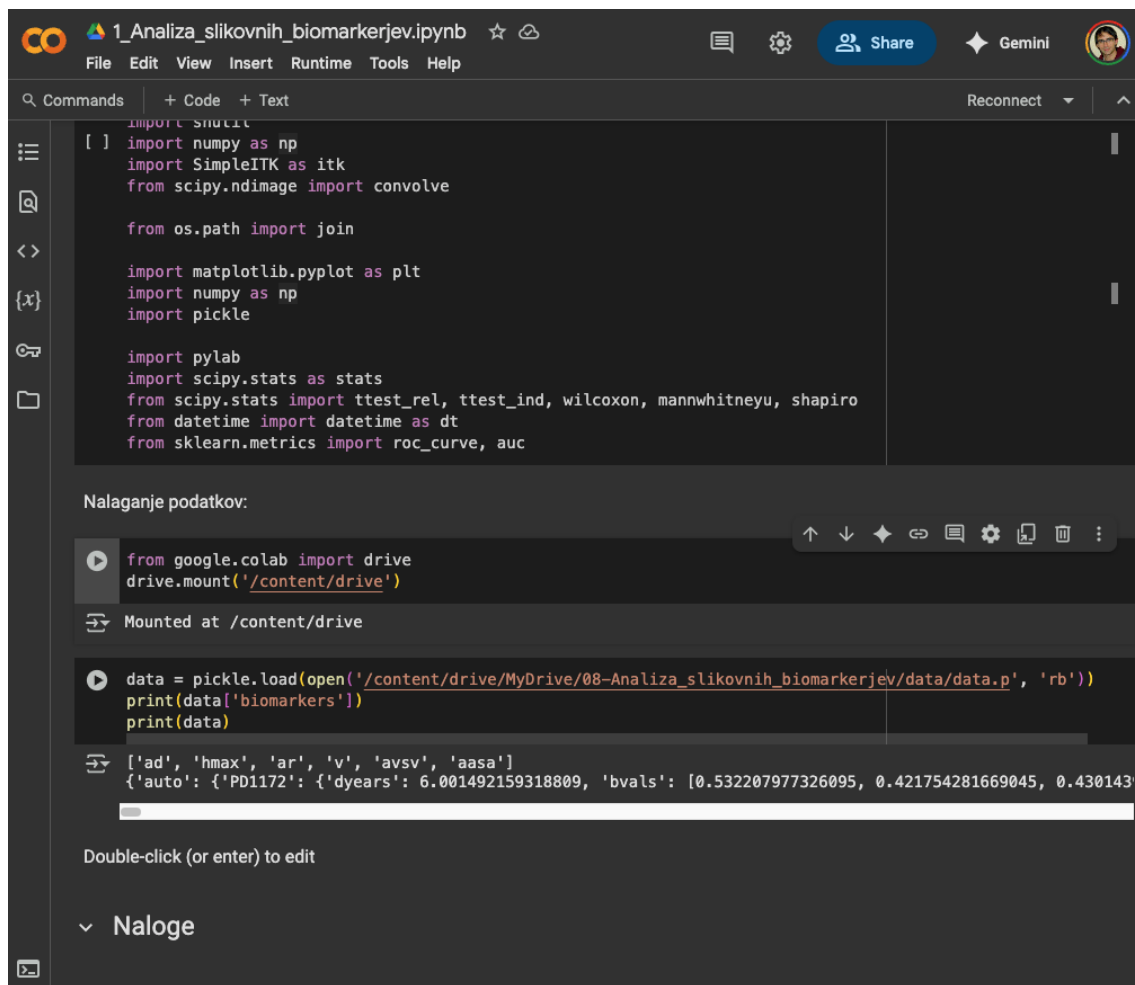
Opomba

: Uporaba gesel namesto SSH ključev ni najbolj varna za produkcijske okolje. Priporočljivo je uporabiti SSH ključe in onemogočiti gesla v produkciji. Poleg tega preverite, ali je vaš požarni zid (ang. *firewall*) nastavljen za dovoljenje izbranega porta (npr. 2222).

Za dodatne informacije obiščite dokumentacijo Dockerja (<https://docs.docker.com>) in OpenSSH (<https://www.openssh.com>).

C.6 Uporaba spletnih razvojnih okolij: Google Colab

Google Colab (<https://colab.research.google.com>) je brezplačno spletno razvojno okolje, ki omogoča uporabo Pythona brez potrebe po lokalni namestitvi, saj se izvaja neposredno v spletnem brskljalniku, kar ga naredi idealnega za začetnike in razvijalce, ki potrebujejo hitre teste ali dostop do naprednih računskih virov. Temelji na Jupyter delovnih zvezkih (za jezik Python imajo končnico `.pynb`) in omogočajo kombiniranje besedila, grafik, tabel in enačb z bloki Python kode (slika C.1).



Slika C.1: Spletno razvojno okolje Google Colab.

V tem podpoglavju bomo pregledali prednosti, navodila za uporabo, primer za računalniški vid ter omejitve tega orodja. Ključne prednosti Google Colab platforme so:

- **Brezplačen dostop do GPU/TPU:** Omogoča hitrejše izvajanje zahtevnejših računskih operacij, kot so globoko učenje.
- **Enostavna delitev notebookov:** Sodelovanje je preprosto prek povezave ali prek Google Drive.
- **Integracija z Google Drive:** Shranjevanje in upravljanje projektov neposredno v oblaku.

Nekatere omejitve platforme so odvisnost od internetne povezave, saj brez interneta ni mogoče dostopati do Colaba ali shranjevati dela. Viri za dolgotrajn računsko izvajanje so omejeni; namreč brezplačni račun ima omejeno količino RAM-a, CPU/GPU časa in shranjevalnega prostora. Dolgotrajno učenje globokih modelov ali obdelava velikih podatkovnih naborov lahko zahtevajo nadgradnjo na plačljive možnosti ali uporabo lokalnega okolja.

C.6.1 Navodila za ustvarjanje in konfiguracijo projekta

Za začetek uporabite Google Colab, obiščite <https://colab.research.google.com> in se prijavite z Google računom. Ustvarite nov delovni zvezek z gumbom “New Notebook”. Osnovna konfiguracija vključuje namestitev potrebnih knjižnic, kot je OpenCV, z uporabo ukaza `!pip install` v celici kode pa lahko namestite druge poljubne Python knjižnice, na primer:

```
!pip install opencv-python matplotlib
```



```
Collecting opencv-python
  Downloading opencv-python-4.5.5.64-cp39-cp39-manylinux1_x86_64.whl (49.4 MB)
    |.....| 49.4 MB 2.3 MB/s
Collecting matplotlib
  Downloading matplotlib-3.5.2-cp39-cp39-manylinux1_x86_64.whl (11.3 MB)
    |.....| 11.3 MB 3.0 MB/s
Installing collected packages: opencv-python, matplotlib
Successfully installed matplotlib-3.5.2 opencv-python-4.5.5.64
```

Po namestitvi dodajte blok kode in preverite verzije:

```
1 import cv2
2 import matplotlib.pyplot as plt
3
4 print("OpenCV Version:", cv2.__version__)
5 print("Matplotlib Version:", plt.__version__)
```

Izhod bo izgledal približno takole:

```
OpenCV Version: 4.5.5.64
Matplotlib Version: 3.5.2
```

C.6.2 Primer uporabe Google Colab

Google Colab je odličen za naloge, kot so nalaganje slik, obdelava z OpenCV in vizualizacija z Matplotlib. Tukaj je primer kode za nalaganje slike iz spletnega mesta, njeno obdelavo in prikaz:

```
1 # Naloži sliko iz URL-ja
2 !wget
   https://fe.uni-lj.si/wp-content/uploads/2024/01/Inf_Laibach-206-scaled.jpg
   -O image.jpg
3
4 # Preberi in obdelaj sliko z OpenCV
5 import cv2
6 import matplotlib.pyplot as plt
7
8 img = cv2.imread('image.jpg')
9 img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB) # Pretvori iz BGR v RGB za
   pravilno prikazovanje
10
11 # Prikaz slike
12 plt.imshow(img_rgb)
13 plt.axis('off') # Skrij osi
14 plt.title('Obdelana slika')
15 plt.show()
```

Ta primer prikazuje, kako naložiti sliko, jo pretvoriti v pravilno barvno shemo (OpenCV privzeto uporablja BGR, medtem ko Matplotlib pričakuje RGB) in jo prikazati. Sliko lahko naložite tudi iz Google Drive z uporabo ustreznih Colab ukazov (npr. `from google.colab import files`), kot je prikazano v sliki C.1.

C.6.3 Namigi za uporabo GPU/TPU:

- V »Runtime« → »Change runtime type« → izberite »GPU«.

- Uporabite `!nvidia-smi` za preverjanje GPU:

```
!nvidia-smi
```

Izhod bo izgledal (zavisi od vaše strojne opreme) približno tako:

NVIDIA-SMI 470.63.01 Driver Version: 470.63.01 CUDA Version: 11.4									
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile Uncorr. ECC				
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.			
						MIG M.			
0	NVIDIA GeForce ...	Off	00000000:01:00.0	Off	N/A	N/A			
0%	45C	P5	12W / 250W	0MiB / 11178MiB	0%	Default			
						N/A			
1	NVIDIA GeForce ...	Off	00000000:02:00.0	Off	N/A	N/A			
34%	48C	P0	54W / 250W	0MiB / 11019MiB	0%	Default			
						N/A			
Processes:									
GPU	GI	CI	PID	Type	Process name	GPU Memory			
ID	ID	ID					Usage		
No running processes found									

C.7 Lokalno razvojno okolje: Visual Studio Code

Visual Studio Code (VS Code) je priljubljen, lahkoten urejevalnik kode s podporo za številne programske jezike, vključno s Pythonom (vmesnik na sliki C.2). Ponuja zmogljive funkcije, kot so avtomatsko dokončevanje kode, razhroščevanje (ang. *debugging*) in integracija z orodji za upravljanje okolij. V tem podpoglavju bomo podrobno opisali namestitev in konfiguracijo VS Code, uporabo Python razširitve, upravljanje knjižnic z Conda ter uporabo Docker kontejnerjev za izolacijo in reproducibilnost projektov.

C.7.1 Namestitev in osnovna konfiguracija Visual Studio Code

VS Code je na voljo za Windows, macOS in Linux ter se prenese z uradne strani <https://code.visualstudio.com/>. Sledite tem korakom za namestitev:

- **Windows:**

- Prenesite namestitveni program z <https://code.visualstudio.com/>
- Zaženite namestitev z privzetimi nastavitvami
- Med namestitvijo izberite možnost dodajanja v sistemsko pot (PATH), če je na voljo

- **macOS:**

```
brew install --cask visual-studio-code
```

- **Linux (Ubuntu/Debian):**

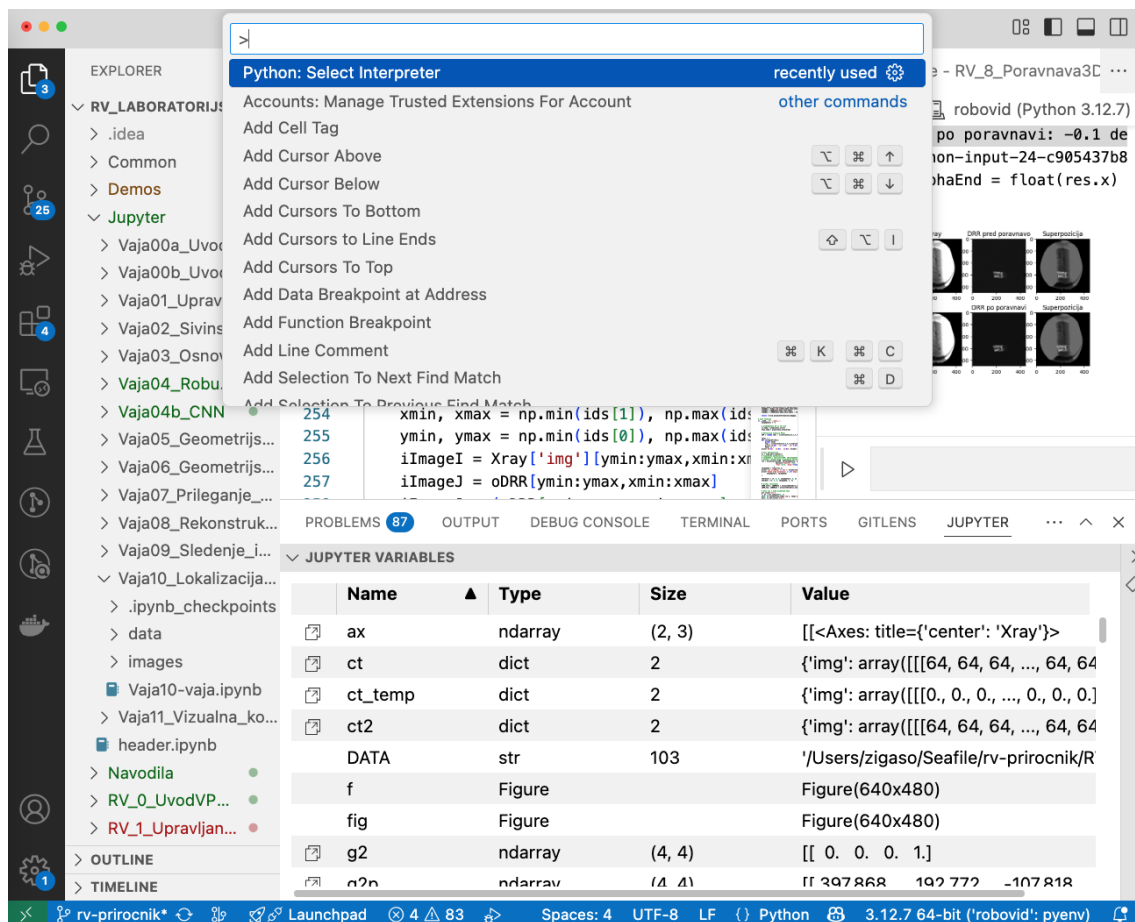
```
sudo apt update
sudo apt install code
```

Preverite delovanje z odpiranjem terminala znotraj VS Code ("View > Terminal") in vnesite:

```
$ code --version
1.82.0
```

Če ukaz deluje, je VS Code pravilno nameščen. Nadaljujte z nameščanjem Python razširitev:

1. Odprite VS Code in pojdite na "tržnico razširitev" (ang. *Extensions Marketplace*; bližnjica `Ctrl+Shift+X`)
2. Poiščite in namestite "Python" razširitev od Microsoft s klikom na gumb "Install"
3. Priporočene dodatne razširitve:
 - Pylance (za napredno dokončevanje kode)
 - Python Indent (za pravilno oblikovanje)
 - Jupyter (za delo z notebooki)



Slika C.2: Grafični vmesnik Visual Studio Code (VS Code).

Po namestitvi razširitve odprite novo Python datoteko (.py) in začnite pisati kodo. Razširitev bo samodejno predlagala dokončanje (npr. za funkcije iz numpy). Za linting namestite v terminalu v VS Code orodje, kot je `flake8`, z:

```
$ pip install flake8
```

V nastavitvah VS Code omogočite linting z dodajanjem v `settings.json`:

```
{
  "python.linting.enabled": true,
  "python.linting.flake8Enabled": true
}
```

Za razhroščevanje nastavite kontrolne točke (ang. *breakpoints*) v kodi in uporabite gumb “Run and Debug” v levem meniju VS Code. Po zaustavitvi v kontrolni točki lahko preverite vrednosti spremenljivk ali po potrebi tudi spremenite.

C.7.2 Uporaba lokalne Python distribucije

Predpostavimo, da ste že namestili Python preko uradne strani (<https://www.python.org/downloads/>) ali Anaconda. VS Code mora prepoznati to distribucijo za zaganjanje Python programov:

- Odprite VS Code in ustvarite novo mapo za projekt (“File > Open Folder”).

- Odprite paletu ukazov (Ctrl+Shift+P ali Cmd+Shift+P na macOS) in vnesite “Python: Select Interpreter” (kot na sliki C.2). Izberite lokalno nameščeno verzijo Pythona, npr.:

```
Python 3.9.7 64-bit ('base': conda)
```

Če Python ni zaznana, preverite, ali je pot do `python.exe` (Windows) ali `python3` (Linux/macOS) dodana v sistemsko spremenljivko `PATH`. Na Windows lahko to nastavite ročno v Nastavitvah sistema, na Linux/macOS pa z urejanjem `~/.bashrc` ali `~/.zshrc`:

```
$ echo 'export PATH="$HOME/.pyenv/bin:$PATH"' >> ~/.bashrc
$ source ~/.bashrc
```

Preverite delovanje v terminalu z:

```
$ python --version
Python 3.9.7
```

- Nastavite konfiguracijo za poganjanje kode in razhroščevanje z ustvarjanjem `launch.json` (preko palete ukazov “Debug: Open launch.json”). Primer:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Python: Current File",
      "type": "python",
      "request": "launch",
      "program": "${file}",
      "console": "integratedTerminal"
    }
  ]
}
```

C.7.3 Uporaba lokalnega Docker kontejnerja

VS Code omogoča delo z lokalnimi in oddaljenimi (*takimi, ki tečejo na drugih računalnikih*) Docker kontejnerji preko SSH.

Predpogoj je, da imate nameščen in konfiguriran Docker, zagnan Docker kontejner s SSH strežnikom in omogočen dostop do SSH porta v kontejnerju (kot je opisano v poglavju C.5.1) ter VS Code z ustreznimi razširitvami. Priporočamo, da imate:

- Nameščen Docker Desktop ali Docker CE na vašem sistemu (<https://www.docker.com>).
- Konfiguriran Dockerfile z SSH podporo, kot je prikazano prej (npr. z `openssh-server` in izpostavljenim portom 22).
- Nameščen VS Code z razširitvijo “Remote - SSH” iz tržnice razširitev.

Nato odprite paletu ukazov (Ctrl+Shift+P ali Cmd+Shift+P na macOS) in vnesite “Remote-SSH: Connect to Host”. To bo omogočilo konfiguracijo SSH povezave. Preverite delovanje razširitve z odpiranjem terminala znotraj VS Code:

```
$ code --install-extension ms-vscode-remote.remote-ssh
Installing extension 'ms-vscode-remote.remote-ssh' v1.93.0...
Extension 'ms-vscode-remote.remote-ssh' v1.93.0 was successfully installed.
```

Predpostavimo, da imate že zagnan Docker kontejner z SSH v odcepljenem načinu in izpostavlje port 2222. Nato konfigurirajte SSH v VS Code:

1. Ustvarite ali uredite datoteko konfiguracije SSH na vašem računalniku. Običajno se nahaja v `~/.ssh/config` (Linux/macOS) ali `C:\Users\YourUser\.ssh\config` (Windows). Dodajte naslednjo vsebino:

```
Host docker-container
  HostName localhost
  User developer
  Port 2222
  IdentityFile ~/.ssh/id_rsa
```

Če še nimate SSH ključa, ga ustvarite v terminalu z:

```
$ ssh-keygen -t rsa -b 4096 -f ~/.ssh/id_rsa
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in ~/.ssh/id_rsa
Your public key has been saved in ~/.ssh/id_rsa.pub
```

Kopirajte javni ključ (`~/.ssh/id_rsa.pub`) v kontejner, če ga še niste (lahko uporabite `docker cp` ali ročno med zagonom).

2. V VS Code odprite paletu ukazov in izberite “Remote-SSH: Connect to Host”. Izberite “docker-container” iz konfiguracije. VS Code bo vzpostavil povezavo in odprl novo okno, povezano z vašim Docker kontejnerjem.

Ko ste povezani preko SSH, lahko v VS Code urejate in izvajate Python kodo, kot bi bili na lokalnem sistemu. Lahko namestite dodatne knjižnice z `pip` ali `conda` znotraj kontejnerja. Lahko poganjate in razhroščujete kodo z nastavitvijo kontrolnih točk in uporabo integriranega razhroščevalnika v VS Code. Za dodatne informacije obiščite dokumentacijo VS Code (<https://code.visualstudio.com/docs/remote/ssh>) in Dockerja (<https://docs.docker.com>).



D. Vodenje različic kode z Git

D.1 Kaj je Git in zakaj ga uporabljamo?

Git je razširljiv, hiter in razpršen **sistem za upravljanje in vodenje različic** (angl. *Version Control System, VCS*), ki ga je leta 2005 ustvaril Linus Torvalds za razvoj jedra Linux. Omogoča sledenje spremembam v izvorni kodi, usklajeno sodelovanje več razvijalcev in enostavno obnavljanje starejših različic projekta. Tu je nekaj ključnih namenov uporabe Gita:

- **Sledenje spremembam:** Git beleži vse spremembe v datotekah, kar omogoča primerjavo ali povrnitev na prejšnje različice.
- **Sodelovanje:** Več razvijalcev lahko hkrati dela na istem projektu brez prepričkov (branches, merge).
- **Odvečnost:** Ker je razpršen, vsak razvijalec ima celotno zgodovino projekta – ni odvisen od centralnega strežnika.
- **Integracija s platformami:** Git se pogosto uporablja s storitvami, kot so GitHub, GitLab in Bitbucket, za skupno razvijanje kode.
- **Hitrost in učinkovitost:** Git je optimiziran za hitro delo, tudi z velikimi projekti.

Git se danes uporablja v skoraj vseh večjih programerskih projektih, od odprtokodne programske opreme do komercialnih rešitev. Njegova fleksibilnost in zmogljivost sta ključna razloga za njegovo široko sprejetost.

D.2 Namestitev Gita

Git je razpoložljiv za vse glavne operacijske sisteme. Spodaj so navodila za namestitev na različne platforme.

D.2.1 Namestitev na Windows

1. Obiščite uradno spletno stran <https://git-scm.com/>.
2. Prenesite namestitveni program za Windows (‘.exe’ datoteko).
3. Zaženite namestitveni program in sledite navodilom (priporočljivo je ohraniti privzete nastavitve).

D.2.2 Namestitev na macOS

1. Če še nimate Homebrew, ga namestite z ukazom:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/  
↳ Homebrew/install/HEAD/install.sh)"
```

2. Nato namestite Git z ukazom:

```
brew install git
```

D.2.3 Namestitev na Linux

1. Za distribucije, ki uporabljajo apt (npr. Ubuntu, Debian), uporabite:

```
sudo apt update  
sudo apt install git
```

2. Za distribucije z yum (npr. Fedora, CentOS) uporabite:

```
sudo yum install git
```

D.2.4 Preverjanje namestitve

Ne glede na operacijski sistem lahko po namestitvi preverite, če Git deluje pravilno, z ukazom:

```
git --version
```

Če je Git pravilno nameščen, se izpiše različica (npr. `git version 2.40.1`).

D.3 Osnovne nastavitve

Preden začnemo uporabljati orodje Git, je potrebno opraviti osnovno nastavitve, ki omogoča, da Git ve, kdo ste in kako naj deluje. V tem poglavju bomo pregledali osnovne nastavitve, kot so nastavitve uporabniškega imena in e-pošte, privzeto nastavitve urejevalnika ter prikaz trenutne nastavitve.

D.3.1 Nastavitev uporabniškega imena in e-pošte

Ena izmed prvih stvari, ki jih morate nastaviti pri uporabi Gita, je vaše uporabniško ime in e-poštni naslov. Ti podatki se uporabljajo za označevanje sprememb, ki jih vnesete v repozitorij. Nastavitev je globalna in se izvede z ukazoma `git config -global user.name` za uporabniško ime in `git config -global user.email` za e-poštni naslov. Primer:

```
\$ git config --global user.name "Vaše Ime"  
\$ git config --global user.email "vaša.email@primer.com"
```

Ko enkrat nastavite te vrednosti, jih Git uporabi za vse vaše repozitorije, razen če jih posebej prilagodite za posamezen repozitorij brez globalne nastavitve.

D.3.2 Privzeto nastavljanje urejevalnika

Git pogosto zahteva uporabo urejevalnika besedil, na primer pri pisanju sporočil ob potrditvi sprememb (ang. *commit messages*) ali urejanju nastavitvev. Privzeti urejevalnik lahko nastavite z ukazom `git config -global core.editor`. Pogosto uporabljeni urejevalniki so na primer Vim, Nano ali Visual Studio Code. Primer nastavitve za uporabo Vima:

```
\$ git config --global core.editor "vim"
```

Če želite uporabiti drug urejevalnik, ga enostavno zamenjate z ustreznim imenom programa. To nastavitve je priporočljivo opraviti takoj, saj olajša delo z Gitom.

D.3.3 Prikaz konfiguracije

Za preverjanje trenutnih nastavitev lahko uporabite ukaz `git config -list`, ki prikaže vse konfiguracijske nastavitve, ki jih Git uporablja. Primer izhoda:

```
\$ git config --list
user.name=Vaše Ime
user.email=vaša.email@primer.com
core.editor=vim
...
```

Ta ukaz je koristen, če želite preveriti, ali so vaše nastavitve pravilno shranjene ali če potrebujete pregled nad trenutno konfiguracijo. Če katera od nastavitev manjka, jo lahko kadar koli dodate ali spremenite z ustreznim ukazom `git config`.

S temi osnovnimi nastavitvami ste pripravljeni na nadaljnjo uporabo Gita. Naslednji koraki običajno vključujejo ustvarjanje repozitorijev in upravljanje verzij vaših projektov.

D.4 Ustvarjanje in kloniranje repozitorija

Ena izmed osnovnih operacij pri uporabi Gita je ustvarjanje novih repozitorijev ali kloniranje obstoječih. V tem poglavju bomo pregledali, kako inicializirati nov repozitorij in kako klonirati že obstoječ repozitorij iz oddaljenega strežnika, kot je na primer GitHub.

D.4.1 Inicializacija novega repozitorija

Če želite začeti nov projekt pod verzijsko kontrolo z Gitom, morate inicializirati nov repozitorij. To naredite z ukazom `git init`, ki ustvari novo Git mapo (direktorij) v trenutni mapi. Primer:

```
\$ git init
Initialized empty Git repository in /path/to/your/project/.git/
```

Ko izvedete ta ukaz, Git ustvari skrito mapo `.git`, ki vsebuje vse potrebne datoteke za sledenje spremembam v vašem projektu. Po inicializaciji lahko začnete dodajati datoteke, jih zavezati (commit) in upravljati verzije.

D.4.2 Kloniranje obstoječega repozitorija

Če želite delati z že obstoječim repozitorijem, na primer tistim, ki je shranjen na platformi kot je GitHub, Bitbucket ali GitLab, lahko uporabite ukaz `git clone <url>`. Ta ukaz naredi kopijo celotnega repozitorija, vključno z zgodovino sprememb, na vaš lokalni računalnik. Primer:

```
\$ git clone https://github.com/uporabnik/ime-repozitorija.git
Cloning into 'ime-repozitorija'...
remote: Enumerating objects: 100, done.
remote: Counting objects: 100% (100/100), done.
remote: Compressing objects: 100% (50/50), done.
Receiving objects: 100% (100/100), 1.23 MiB | 1.23 MiB/s, done.
Resolving deltas: 100% (60/60), done.
```

V tem primeru nadomestite `https://github.com/uporabnik/ime-repozitorija.git` z dejanskim URL-jem repozitorija, ki ga želite klonirati. Po uspešnem kloniranju lahko začnete delati z datotekami in jih po potrebi spreminjati, zavezovati in pošiljati nazaj na oddaljeni strežnik.

S temi koraki ste pripravljeni na upravljanje svojih projektov z Gitom, bodisi z ustvarjanjem novih repozitorijev bodisi z delom na obstoječih. Naslednji koraki običajno vključujejo dodajanje in posodabljanje datotek ter sodelovanje z drugimi razvijalci.

D.5 Osnovni Git ukazi

Osnovni Git ukazi vam omogočajo upravljanje sprememb v vašem projektu. Pregledali bomo ukaze za preverjanje statusa datotek, dodajanje sprememb v “staging” (priprava spremembe) območje, ustvarjanje “commitov” (potrditev spremembe) in ogled zgodovine sprememb.

D.5.1 Preverjanje statusa datotek

Za preverjanje trenutnega stanja vašega repozitorija, vključno z neizvedenimi spremembami, novimi datotekami in datotekami, pripravljenimi za potrditev spremembe (“commit”), uporabite ukaz `git status`. Ta ukaz prikaže podrobnosti o tem, kaj se je spremenilo od zadnje potrditve spremembe. Primer:

```
\$ git commit
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   dokumentacija.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    nova_datoteka.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Ukaz `git status` je koristen za hitro oceno, kaj morate nadalje narediti.

D.5.2 Dodajanje sprememb v pripravo

Preden lahko spremembe potrdite (commit), jih morate dodati v stanje pripravljenosti (“staging”). To naredite z ukazom `git add <file>` za posamezno datoteko ali `git add .` za dodajanje vseh sprememb v trenutni mapi. Primeri:

```
\$ git add dokumentacija.txt    # Dodajanje posamezne datoteke
\$ git add .                    # Dodajanje vseh sprememb v trenutni mapi
```

Ko so datoteke v stanju pripravljenosti, jih v naslednjem koraku lahko potrdite – ustvarite “commit”.

D.5.3 Potrjevanje sprememb

Za potrditev in shranjevanje sprememb v zgodovino repozitorija uporabite ukaz `git commit -m "Sporočilo"`, kjer "Sporočilo" opisuje, katere spremembe ste naredili. Primer:

```
\$ git commit -m "Posodobljena dokumentacija in dodana nova datoteka"
[main 7f3c5b6] Posodobljena dokumentacija in dodana nova datoteka
 2 files changed, 15 insertions(+)
 create mode 100644 nova_datoteka.txt
```

Sporočilo potrditve spremembe mora biti jasno in opisno, da olajša razumevanje zgodovine projekta.

D.5.4 Ogled zgodovine sprememb

Za pregled zgodovine vseh sprememb v repozitoriju uporabite ukaz `git log`. Ta ukaz prikaže seznam commitov z njihovimi ID-ji, avtorji, datumi in sporočili. Primer:

```
\$ git log
commit 7f3c5b6a9d8e7f6b5c4d3e2a1b0 (HEAD -> main, origin/main)
Author: Vaše Ime <vaša.email@primer.com>
Date:   Wed Oct 16 14:30:22 2023 +0200
```

Posodobljena dokumentacija in dodana nova datoteka

```
commit 5e4d3c2b9a8f7e6d5c4b3a2 (origin/develop, develop)
Author: Druga Oseba <druga.oseba@primer.com>
Date:   Tue Oct 15 10:15:00 2023 +0200
```

Osnovna struktura projekta

Ukaz `git log` je koristen za sledenje spremembam in razumevanje razvoja projekta.

S temi osnovnimi ukazi lahko učinkovito upravljate svoje projekte z Gitom. Naslednji koraki lahko vključujejo delo z vejami, združevanje sprememb in sodelovanje z drugimi razvijalci.

D.6 Uporaba vejitev

Vejitve (ang. *branching*) so eden izmed močnejših funkcionalnosti Gita, ki omogoča vzporedno delo na različnih delih projekta brez vpliva na glavno vejo (npr. `main` ali `master`). V tem podpoglavju bomo pregledali, kako ustvarjati in preklapljati med vejami, kako združevati veje ter kako reševati morebitne konflikte.

D.6.1 Ustvarjanje in preklapljanje vej

Za upravljanje vej v Gitu uporabljamo ukaze, kot so `git branch`, `git checkout` in `git switch`. Z ukazom `git branch` lahko ustvarite novo vejo ali si ogledate obstoječe veje. Za preklapljanje med vejami pa lahko uporabite `git checkout` ali novejši `git switch`. Primeri:

```
\$ git branch nova-vej          # Ustvari novo vejo z imenom "nova-veja"
\$ git branch                  # Prikaže vse obstoječe veje (trenutna je
  ↳ označena z *)
  main
  * nova-vej

\$ git checkout nova-vej        # Preklopi na vejo "nova-veja"
Switched to branch 'nova-vej'

# Ali z novejšim ukazom:
\$ git switch nova-vej          # Preklopi na vejo "nova-veja"
Switched to branch 'nova-vej'
```

Ukaz `git branch` brez argumentov prikaže seznam vseh vej, medtem ko `git checkout` ali `git switch` omogočata delo na izbrani veji.

D.6.2 Združevanje vej

Ko končate delo na veji in želite njene spremembe vključiti v glavno vejo (npr. `main`), uporabite ukaz `git merge`. Najprej se preklopite na vejo, v katero želite združiti spremembe, nato pa izvedete ukaz `merge`. Primer:

```
\$ git checkout main           # Preklopi na glavno vejo
Switched to branch 'main'
```

```
\$ git merge nova-veja          # Združi vejo "nova-veja" v "main"
Updating 7f3c5b6..a1b2c3d
Fast-forward
 dokumentacija.txt | 2 ++
 1 file changed, 2 insertions(+)
```

Če ni konfliktov, Git avtomatsko združi veji. V nasprotnem primeru boste morali ročno reševati konflikte.

D.6.3 Reševanje konfliktov

Konflikti se pojavijo, ko dve veji spremenita isto vrstico v isti datoteki na različne načine. Ko Git zazna konflikt med združevanjem (`git merge`), vas obvesti in datoteke, ki vsebujejo konflikte, morate ročno urediti. Primer konflikta in njegove rešitve:

Predstavljajmo si, da imate datoteko `dokumentacija.txt` z naslednjim konfliktom po poskusu združitve:

```
\$ git merge nova-veja
Auto-merging dokumentacija.txt
CONFLICT (content): Merge conflict in dokumentacija.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Odprite datoteko `dokumentacija.txt` in poiščite konfliktne dele, ki izgledajo takole:

```
<<<<<<< HEAD
To je verzija iz glavne veje.
=====
To je verzija iz veje nova-veja.
>>>>>>> nova-veja
```

Ročno uredite datoteko, da odpravite konflikt, na primer v `dokumentacija.txt` napišete:

```
To je združena verzija obeh sprememb.
```

Ko konflikt rešite, dodajte popravljeno datoteko v staging z `git add` in dokončajte postopek združevanja z novo potrditvijo sprememb:

```
\$ git add dokumentacija.txt
\$ git commit
[main 123abc4] Merge branch 'nova-veja'
```

Git bo samodejno ustvaril sporočilo commita, ki opisuje združevanje. Z uporabo vej lahko učinkovito ločujete delo na različnih funkcijah ali popravkih, hkrati pa ohranjate stabilnost glavne veje projekta. Naslednji koraki lahko vključujejo oddaljeno shranjevanje vej ali sodelovanje z drugimi razvijalci.

D.7 Povezava z oddaljenim repozitorijem

Za sodelovanje z drugimi razvijalci ali shranjevanje vašega projekta na oddaljenem strežniku, kot je GitHub, GitLab ali Bitbucket, morate vzpostaviti povezavo med vašim lokalnim repozitorijem in oddaljenim repozitorijem. V tem podpoglavju bomo pregledali, kako dodati oddaljeni repozitorij, poslati spremembe (`git push`) in pridobiti spremembe (`git pull`) iz njega.

D.7.1 Dodajanje oddaljenega repozitorija

Za povezavo vašega lokalnega repozitorija z oddaljenim repozitorijem uporabite ukaz `git remote add origin <url>`, kjer je `origin` privzeto ime za oddaljeni repozitorij, `<url>` pa naslov oddaljenega repozitorija (npr. HTTPS ali SSH URL). Primer:

```
\$ git remote add origin https://github.com/uporabnik/ime-repozitorija.git
```

Po tem koraku lahko vaš lokalni repozitorij komunicira z oddaljenim. Če želite preveriti, katere oddaljene repozitorije so konfigurirani, uporabite:

```
\$ git remote -v
origin https://github.com/uporabnik/ime-repozitorija.git (fetch)
origin https://github.com/uporabnik/ime-repozitorija.git (push)
```

D.7.2 Pošiljanje sprememb

Ko ste pripravljeni deliti svoje lokalne spremembe z oddaljenim repozitorijem, uporabite ukaz `git push`. Ta ukaz pošlje vaše potrjene spremembe na oddaljeni repozitorij, običajno na vejo, kot je `main` ali `master`. Primer:

```
\$ git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (5/5), 623 bytes | 623.00 KiB/s, done.
Total 5 (delta 1), reused 0 (delta 0), pack-reused 0
To https://github.com/uporabnik/ime-repozitorija.git
 7f3c5b6..a1b2c3d main -> main
```

Če pošiljanje sprememb uspe, so vaše spremembe zdaj na voljo na oddaljenem strežniku. Prvič morda potrebujete poverila (uporabniško ime in geslo ali SSH ključ), odvisno od konfiguracije.

D.7.3 Pridobivanje sprememb

Če želite posodobiti svoj lokalni repozitorij z najnovejšimi spremembami iz oddaljenega repozitorija, uporabite ukaz `git pull`. Ta ukaz prenese in avtomatsko združi spremembe iz oddaljenega repozitorija v vašo trenutno vejo. Primer:

```
\$ git pull origin main
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 2 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (2/2), 211 bytes | 211.00 KiB/s, done.
From https://github.com/uporabnik/ime-repozitorija.git
  a1b2c3d..b2c3d4e main -> origin/main
Updating a1b2c3d..b2c3d4e
Fast-forward
 dokumentacija.txt | 1 +
 1 file changed, 1 insertion(+)
```

Če pride do konfliktov med združevanjem, jih morate ročno rešiti, podobno kot pri združevanju vej (glej poglavje D.6.3). S temi koraki lahko učinkovito upravljate sinhronizacijo med vašim lokalnim in oddaljenim repozitorijem, kar je ključno za timsko delo in varnostno kopiranje vašega projekta.

D.8 Pogoste napake in koristni namigi

Med delom z Gitom se lahko zgodi, da naredite napake ali potrebujete dodatne orodja za boljše upravljanje projekta. V tem podpoglavju bomo pregledali, kako razveljaviti spremembe, kako

pridobiti izgubljene potrjene spremembe in kako ignorirati nepotrebne datoteke z uporabo datoteke `.gitignore`.

D.8.1 Razveljavljanje sprememb

Pogosto se zgodi, da želite razveljaviti spremembe, ki ste jih naredili v delovni mapi ali pri pripravi datotek (*staging*). Za to lahko uporabite ukaze, kot sta `git checkout - <file>` in `git reset`. Primeri:

- Če želite zavreči spremembe v določeni datoteki, ki še niso bile dodane v stanje pripravljenosti:

```
\$ git checkout -- dokumentacija.txt
```

To vrne datoteko `dokumentacija.txt` na stanje zadnjega commita.

- Če želite odstraniti datoteke iz stanja pripravljenosti, ne da bi spremenili njihovo vsebino v delovni mapi, uporabite:

```
\$ git reset dokumentacija.txt
Unstaged changes after reset:
M    dokumentacija.txt
```

Če ste že naredili potrditev spremembe (*commit*), ki jo želite razveljaviti, lahko uporabite `git reset` z dodatnimi možnostmi, kot je `-soft` ali `-hard`, odvisno od tega, ali želite ohraniti spremembe ali ne.

D.8.2 Pridobivanje izgubljenih potrjenih sprememb

Včasih lahko po pomoti izbrišete veje ali potrjene spremembe. Za obnovitev izgubljenih potrjenih sprememb lahko uporabite `git reflog`, ki beleži vse reference na prejšnje potrjene spremembe, tudi če so bile izbrisane. Primer:

```
\$ git reflog
7f3c5b6 HEAD@{0}: reset: moving to HEAD^
a1b2c3d HEAD@{1}: commit: Posodobljena dokumentacija
5e4d3c2 HEAD@{2}: checkout: moving to main
```

Ko najdete ID potrjene spremembe (npr. `a1b2c3d`), jo lahko obnovite z:

```
\$ git checkout a1b2c3d
Note: switching to 'a1b2c3d'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.

Nato lahko ustvarite novo vejo iz te potrjene spremembe, če želite ohraniti te spremembe.

D.8.3 Ignoriranje datotek

Za preprečevanje, da bi Git sledil določenim datotekam (npr. začasnim datotekam, dnevnikom ali konfiguracijskim datotekam, za katere ne želite shranjevati različic), uporabite datoteko `.gitignore`. V njo vnesete vzorce datotek ali map, ki jih želite ignorirati. Primer datoteke `.gitignore`:

```
# Ignoriraj vse log datoteke
*.log

# Ignoriraj specifično mapo
temp/
```



```
# Ignoriraj Python bytecode datoteke
*.pyc
```

Datoteko `.gitignore` ustvarite v korenu vašega repozitorija in jo dodate v vodenje različic z ukazom `git add .gitignore` in `git commit`. Od tega trenutka Git ne bo več sledil navedenim datotekam ali mapam.

S temi nasveti lahko hitreje rešujete pogoste težave in izboljšate svoje delo z Gitom.

D.9 Zaključek

D.10 Dobra praksa priprave Git repozitorija

Pri ustvarjanju in vzdrževanju Git repozitorija je pomembno upoštevati dobre prakse, ki olajšajo delo vam in drugim razvijalcem. Tukaj je pregled ključnih elementov, kot so README, oblikovanje, struktura kode in podatkov ter licenca.

- **README datoteka:** Vsak repozitorij bi moral vsebovati datoteko README (najpogosteje v formatu README.md za Markdown), ki služi kot uvod v projekt. V njej navedite cilj projekta, kako ga namestiti, uporabljati in prispevati k njemu. Primer dobra praksa je, da vključite naslove, slike, povezave in primere kode.
- **Oblikovanje:** Kode in dokumentacije se držite konsistentnega stila. Uporabite orodja, kot so linterji ali formatirji (npr. Prettier za JavaScript ali Black za Python), da zagotovite, da je koda berljiva in skladna s standardi. Za jezik Python se držite standarda PEP 8¹ (ang. *Python Enhancement Proposal* 8). Dodajte tudi datoteke, kot so `.editorconfig` ali `.prettierrc`, za avtomatizacijo oblikovanja.
- **Struktura kode in podatkov:** Organizirajte datoteke v logične mape, npr. `src/` za izvorno kodo, `tests/` za teste, `docs/` za dokumentacijo in `data/` za podatke. Izognite se kaotičnemu dodajanju datotek v koren repozitorija. Če projekt vsebuje velike datoteke ali binarne podatke, razmislite o uporabi orodij, kot je Git LFS (ang. *Large File Storage*).
- **Razvojno okolje:** podajte navodila za pripravo razvojnega okolja (različica Python, tip razvojnega okolja) in vključite `requirements.txt` v svoj Git repozitorij, omogočite drugim, da hitro nastavijo razvojno okolje z enakimi odvisnostmi. Primer vsebine datoteke za izvedbo vaj v tem priročniku je `requirements.txt`.
- **Licenca:** Vsak repozitorij bi moral imeti jasno določeno licenco, ki določa, kako lahko drugi uporabljajo, spreminjajo ali distribuirajo vašo kodo. Pogoste izbire so MIT, GPL ali Apache licenca. Datoteko licence (npr. LICENSE) dodajte v koren repozitorija in jo omenite v README. Preverite katera licenca je primerna za vaš projekt na <https://choosealicense.com/> ali se pozanimajte o posameznih licencah na <https://tldrlegal.com/>

S temi praksami zagotovite, da je vaš Git repozitorij profesionalen, enostaven za uporabo in privlačen za potencialne sodelavce ali uporabnike. Naslednji koraki lahko vključujejo redno posodabljanje dokumentacije in preverjanje skladnosti s temi standardi.

D.11 Zaključek

V tem poglavju smo pregledali osnovne koncepte in ukaze orodja Git, ki so ključni za učinkovito upravljanje verzij programske kode in drugih projektov. Git ni le orodje, temveč tudi temeljni del sodobnega razvojnega procesa, ki razvijalcem omogoča sodelovanje, sledenje spremembam in vzdrževanje stabilnosti projektov. V tem zaključku bomo povzeli, zakaj je Git ključen za razvijalce, ter predlagali nadaljnje vire za učenje.

¹ PEP 8: <https://www.python.org/dev/peps/pep-0008/>

D.11.1 Zakaj je Git ključen za razvijalce?

Git je postal standardno orodje v svetu programske razvojne industrije zaradi svojih številnih prednosti:

- **Verzijska kontrola:** Git omogoča sledenje vsaki spremembi v kodu, kar olajša razumevanje, kako se je projekt razvijal skozi čas. S pomočjo ukazov, kot so `git log` in `git diff`, lahko razvijalci hitro identificirajo spremembe in njihove avtorje.
- **Sodelovanje:** Z uporabo vej (`git branch`, `git merge`) in oddaljenih repozitorijev (`git push`, `git pull`) lahko več razvijalcev istočasno dela na istem projektu brez konfliktov, kar je ključno za timsko delo.
- **Fleksibilnost in obnovitev:** Napake so del razvoja, vendar Git omogoča njihovo hitro popravljanje z ukazi, kot so `git reset`, `git checkout` in `git reflog`, kar zmanjša tveganje izgube dela.
- **Široka uporaba:** Platforme, kot so GitHub, GitLab in Bitbucket, temeljijo na Gitu, kar pomeni, da je znanje Gita nujno za sodelovanje v odprtokodnih projektih in profesionalnem okolju.

Skratka, Git je nepogrešljiv, ker zagotavlja strukturo, varnost in učinkovitost pri upravljanju kode, kar razvijalcem omogoča osredotočanje na ustvarjanje, namesto na ročno upravljanje sprememb.

D.11.2 Nadaljnji viri za učenje

Če želite poglobiti svoje znanje o Gitu, je na voljo veliko virov, ki pokrivajo tako osnove kot napredne tehnike. Tukaj je nekaj priporočil:

- **Pro Git knjiga:** To je brezplačna, poglobljena knjiga, ki jo je napisal Scott Chacon in Ben Straub. Na voljo je na spletu na naslovu <https://git-scm.com/book>. Pokriva vse od osnovnih ukazov, kot je `git init`, do naprednih tehnik, kot je ponastavitev korena repozitorija (`git rebase`).
- **Interaktivne vaje:** Strani, kot so GitHub Learning Lab (<https://lab.github.com/>), ponujajo interaktivne vadbice, kjer lahko praktično vadite ukaze, kot so `git add`, `git commit` in `git push`, v realnem okolju.
- **Video vadbice in tečaji:** Platforme, kot so Coursera, Udemy in YouTube, ponujajo številne tečaje o Gitu, ki vključujejo praktične primere in nasvete od izkušenih razvijalcev.
- **Dokumentacija in forumi:** Uradna stran Gita (<https://git-scm.com/>) in skupnosti, kot so Stack Overflow, nudijo podporo in odgovore na specifična vprašanja.

S temi viri lahko nadaljujete učenje in izpopolnite svoje veščine, da bi še bolj izkoristili moč Gita v svojih projektih. Git ni le orodje, temveč tudi veščina, ki se izpopolnjuje z prakso in razumevanjem njegovih zmogljivosti.

Bibliografija

Knjige

- [1] Wilhelm Burger in Mark J. Burge. *Principles of Digital Image Processing: Fundamendal/ Core Techniques*. Springer, 2009 (cit. na str. 20).
- [2] Miran Bürmen. *Uvod v programski jezik Python*. 1. izd. Ljubljana: Založba FE, 2016. ISBN: 978-961-243-318-5. URL: <https://plus.cobiss.net/cobiss/si/sl/bib/287674112> (pridobljeno 7. 4. 2025) (cit. na str. 20, 225).
- [3] Gene H. Golub in Charles F. Van Loan. *Matrix Computations*. 3rd edition. Baltimore: Johns Hopkins University Press, okt. 1996. ISBN: 978-0-8018-5414-9 (cit. na str. 217).
- [4] Richard Hartley in Andrew Zisserman. *Multiple View Geometry in Computer Vision*. English. 2nd edition. Cambridge, UK ; New York: Cambridge University Press, apr. 2004. ISBN: 978-0-521-54051-3 (cit. na str. 178).
- [5] Reinhard Klette. *Concise Computer Vision*. Springer, 2014 (cit. na str. 20).
- [6] Boštjan Likar. *Biomedicinska slikovna informatika in diagnostika*. Založba FE in FRI, 2008 (cit. na str. 20).
- [7] Jan Erik Solem. *Programming Computer Vision with Python: Techniques and Libraries for Imaging and Retrieving Information*. Prva izdaja. Sebastopol, CA: O'Reilly Media, 2012. ISBN: 978-1449316549 (cit. na str. 226).
- [8] Milan Sonka, Vaclav Hlavac in Roger Boyle. *Image Processing, Analysis, and Machine Vision*. 4th edition. Stamford, CT, USA: Cengage Learning, jan. 2014. ISBN: 978-1-133-59360-7 (cit. na str. 215).
- [9] Žiga Špiclin. *Robotski vid: laboratorijske vaje v programskem jeziku Matlab*. Opis vira z dne 5. 5. 2020. Univerza v Ljubljani, Fakulteta za elektrotehniko, 2016. URL: <http://lit.fe.uni-lj.si/gradivo/RV-LabVaje-2016-slo.pdf> (cit. na str. 13).
- [10] Žiga Špiclin. *Robotski vid: predavanja*. Ljubljana: Univerza v Ljubljani, Fakulteta za elektrotehniko, 2019. URL: <https://plus.cobiss.net/cobiss/si/sl/bib/13747203> (pridobljeno 25. 3. 2025) (cit. na str. 20).

- [11] Richard Szeliski. *Computer Vision: Algorithms and Applications, 2nd ed.* Predogled knjige za osebno rabo je dostopen tu: <https://szeliski.org/Book/>. Springer, 2022. URL: <https://link.springer.com/book/10.1007/978-3-030-34372-9> (cit. na str. 20).

Članki

- [12] Alexey Dosovitskiy in dr. »An image is worth 16x16 words: Transformers for image recognition at scale«. V: *arXiv preprint arXiv:2010.11929* (2020) (cit. na str. 78).
- [13] Berthold K. P. Horn, Hugh M. Hilden in Shahriar Negahdaripour. »Closed-form solution of absolute orientation using orthonormal matrices«. V: *JOSA A* 5.7 (jul. 1988). Publisher: Optica Publishing Group, str. 1127–1135. ISSN: 1520-8532. DOI: 10.1364/JOSAA.5.001127. URL: <https://opg.optica.org/josaa/abstract.cfm?uri=josaa-5-7-1127> (pridobljeno 1. 4. 2025) (cit. na str. 220).
- [14] Andriy Myronenko in Xubo Song. »Point Set Registration: Coherent Point Drift«. V: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.12 (dec. 2010). Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence, str. 2262–2275. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2010.46. URL: <https://ieeexplore.ieee.org/document/5432191> (pridobljeno 2. 4. 2025) (cit. na str. 109).
- [15] Shaoqing Ren in dr. »Faster R-CNN: Towards real-time object detection with region proposal networks«. V: *Advances in neural information processing systems* 28 (2015) (cit. na str. 78).

Konferenčni članki

- [16] Nicolas Carion in dr. »End-to-end object detection with transformers«. V: *European conference on computer vision*. Springer, 2020, str. 213–229 (cit. na str. 78).
- [17] Jia Deng in dr. »ImageNet: A large-scale hierarchical image database«. V: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. ISSN: 1063-6919. Jun. 2009, str. 248–255. DOI: 10.1109/CVPR.2009.5206848. URL: <https://ieeexplore.ieee.org/document/5206848> (pridobljeno 27. 3. 2025) (cit. na str. 17, 20, 78).
- [18] Tsung-Yi Lin in dr. »Focal loss for dense object detection«. V: *Proceedings of the IEEE international conference on computer vision*. 2017, str. 2980–2988 (cit. na str. 78).
- [19] Wei Liu in dr. »SSD: Single shot multibox detector«. V: *European conference on computer vision*. Springer, 2016, str. 21–37 (cit. na str. 78).
- [20] Joseph Redmon in dr. »You only look once: Unified, real-time object detection«. V: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, str. 779–788 (cit. na str. 78).
- [21] Mingxing Tan, Ruoming Pang in Quoc V Le. »Efficientdet: Scalable and efficient object detection«. V: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, str. 10781–10790 (cit. na str. 78).

Spletni viri

- [22] Kenneth Joy. *Bresenham's Algorithm*. 1999. URL: <https://www.ercankoclar.com/wp-content/uploads/2016/12/Bresenhams-Algorithm.pdf> (pridobljeno 3. 3. 2025) (cit. na str. 33).
- [23] *Mapa okolja in trajektorija kamere*. URL: <https://cvg.cit.tum.de/research/vslam> (pridobljeno 25. 3. 2025) (cit. na str. 19).

-
- [24] *pointcloud.jpg (2000×850)*. URL: https://cvg.cit.tum.de/_media/research/lslam/pointcloud.jpg (pridobljeno 7. 4. 2025) (cit. na str. 175).
- [25] *Robotski vid*. sl-SI. URL: <https://fe.uni-lj.si/predmeti/robotski-vid/> (pridobljeno 25. 3. 2025) (cit. na str. 20).
- [26] *Sledenje gibanja oseb na ulici*. URL: <https://learningspiral.ai/video-motion-tracking/> (pridobljeno 25. 3. 2025) (cit. na str. 19).
- [27] *Strukturirana svetloba za 3D rekonstrukcijo kipa*. URL: <https://www.instructables.com/Structured-Light-3D-Scanning/> (pridobljeno 25. 3. 2025) (cit. na str. 18).
- [28] TIOBE Software. *TIOBE Index for Programming Languages*. Dostopano 20. oktobra 2023. 2023. URL: <https://www.tiobe.com/tiobe-index/> (pridobljeno 20. 10. 2023) (cit. na str. 225).
- [29] *Vizualno preverjanje kakovosti svedra*. URL: <https://sciotex.com/how-do-vision-inspection-systems-work/> (pridobljeno 25. 3. 2025) (cit. na str. 19).
- [30] *Zaznavanje in pobiranje kosov na tekočem traku*. URL: <https://koldpackindia.in/robotic-quality-inspection-system/> (pridobljeno 25. 3. 2025) (cit. na str. 18).



O avtorju

Žiga Špiclin je leta 2006 diplomiral na področju elektrotehnike na Univerzi v Ljubljani, Fakulteta za elektrotehniko, in leta 2011 na isti ustanovi doktoriral iz elektrotehniških znanosti. Med doktorskim študijem je bil gostujoči raziskovalec na Harvardski Univerzi v Združenih državah Amerike, po doktorskem študiju pa gostujoči raziskovalec na Tehniški univerzi München v Nemčiji. Od leta 2020 je izredni profesor na Univerzi v Ljubljani, Fakulteti za elektrotehniko. Njegovo raziskovalno delo v Laboratoriju za slikovne tehnologije obsega razvoj in vrednotenje algoritmov na področju računalniškega vida in analize medicinskih slik, ter razvoj aplikacij, predvsem na področju magnetnoresonančnih preiskav glave za diagnostiko in prognozo nevrodegenerativnih bolezni, srčno-žilnih obolenj kot so možganske anevrizme in prilagajanja načrtov radioterapije. V znanstveno-raziskovalni skupnosti služi kot član programskega odbora konference MICCAI 2011-, MICCAI PRIME 2018-, bil je član organizacijskega odbora delavnic WBIR 2020 & 2024. Je prejemnik nagrade za najboljši raziskovalni dosežek Univerze v Ljubljani, v 2016 in 2024.

Vizitka

Žiga Špiclin

izr. prof. dr.

- ↪ E-pošta: ziga.spiclin@fe.uni-lj.si
 - ✿ Profilna stran na strani Laboratorija za slikovne tehnologije
 - 📄 Povezava na SICRIS profil
 - in Povezava na LinkedIn profil
 - G Povezava na Google Scholar profil
 - 🏠 Povezava na ORCID profil
 - 📄 Povezava na ResearchGate profil
-

